

Programmes
en Turbo Pascal
et Turbo Assembler
sur PC et PS

Programmation système

Mémoires, disques et fichiers

Livre avec disquette

Guillaume de Brébisson

Programmation système

15 rue Gossin — 92543 MONTROUGE Cedex

Editions P.S.I. est une société du Groupe de la Cité

1990

ISBN : 2-86595-632-6



Programmes
en Turbo Pascal
et Turbo Assembler
sur PC et PS

Programmation système

Mémoires, disques et fichiers

Guillaume de Brébisson

P.S.I

La bibliothèque du programmeur

PC

Récursivité et pointeurs en Turbo Pascal 5.5

par Bernard G. Ragot et Anna Berriegts

Programmation orientée objets
Interface graphique en C++ et Turbo C++

par David Hu et Francis Piérot

et aussi...

Guide P.S.I du développeur sous MS-DOS

par Francis Piérot

Guide P.S.I du programmeur UNIX

par Olivier Daudel

Guide P.S.I du programmeur en C

par Jean-Luc Luczak

La programmation en assembleur
sur PC, PS et compatibles

par Peter Norton et John Socha

La programmation sous OS/2

par Peter Norton et Robert Lafore

A VOTRE SERVICE

CATALOGUES ET "LIVRES MICRO"

Je désire recevoir gratuitement : ❐ votre catalogue général
❐ la revue Livres Micro (avec abonnement gratuit)

VOTRE AVIS NOUS INTERESSE

Pour nous permettre de faire de meilleurs livres, adressez-nous vos critiques et suggestions sur le présent ouvrage.

Titre de l'ouvrage : **Programmation système, Mémoire, disques et fichiers**

Ce livre vous donne-t-il toute satisfaction ? ______________________

Avez-vous des commentaires à formuler ? ______________________

____

____

Avez-vous déjà acquis des livres P.S.I. ? __________ Si oui, lesquels ? __________

____

Qu'en pensez-vous ? ______________________

Où les avez-vous achetés ? ❐ Librairie ❐ Boutique micro
❐ Par correspondance

Votre centre d'intérêt ? ❐ PC (ou compatibles) ❐ Macintosh
❐ Atari ❐ Autre__________

Nom ______________ Prénom ______________ Age ______

Adresse ______________________________________

Code Postal ______________ Ville ______________

Profession ______________________________________

A DECOUPER ET A ENVOYER AUX : **Editions P.S.I., Courrier Lecteur**
15 rue Gossin
92543 Montrouge Cédex

Disquette d'accompagnement

Le livre de programmation que vous avez entre les mains est proposé avec une disquette d'accompagnement. Cette disquette de 360 Ko, au format 5 pouces 1/4, est insérée en fin d'ouvrage et contient les exemples de programmes décrits au fil du livre.

Pour vous, programmeur, elle procure un gain de temps appréciable en vous évitant de taper des centaines de lignes de code. Immédiatement disponible, vous pouvez donc l'utiliser sans attendre.

La description complète de cette disquette figure dans l'annexe intitulée *"Votre disquette d'accompagnement"*, en pages 377, 378, 379 et 380.

Fichiers disponibles

Cette disquette fournit différents fichiers :

- des fichiers de programmes source en Pascal et en Assembleur ;
- des fichiers .OBJ et des fichiers .EXE.

Equipement nécessaire

Pour utiliser cette disquette vous devez disposer :

- d'un ordinateur IBM PC ou *100%* compatible équipé d'au moins 640 Ko de mémoire RAM et d'un disque dur ; d'un lecteur de disquettes 360 Ko ; d'une carte écran EGA.
- du système d'exploitation DOS 3.3 (Microsoft ou IBM) ;
- du compilateur *Turbo Pascal* (version 4 à 6) ou *QuickPASCAL* (version 1) et de *Turbo Assembler* (version 1 ou 2).

Accès à la disquette

A l'aide d'un cutter, veuillez soigneusement découper le haut du rabat de couverture, selon la ligne en pointillés.

Précaution d'emploi

Veuillez dupliquer cette disquette avant toute utilisation et travailler avec la copie.

Service technique

En cas de problème, merci de contacter notre service technique au numéro de téléphone suivant :

(1) 47 40 66 42.

Cette disquette ne peut être vendue séparément de l'ouvrage.

A

Florence,

Frédéric,

et Nicolas.

PREFACE

par John Colibri

Les langages de haut niveau ont pour but d'isoler le programmeur du langage machine. En manipulant des concepts de plus en plus abstraits, il augmente sa productivité en évitant de se noyer dans des détails propres à une machine ou à un Operating System.

Cette vue idyllique ne correspond hélas pas toujours à la réalité. Il est des cas dans lesquels ni le langage, ni l'Operating System ne fournissent la possibilité ou la performance souhaitée. Prenons, par exemple, le cas de la liaison série RS 232 C : WRITE (AUX, "BONJOUR"); fonctionnera correctement, pour peu que vous ayez pensé à initialiser les paramètres de la communication avec la commande MODE du DOS. Mais, sur un PC à 4 Mhz vous ne dépasserez pas 2 400 bauds. Il faut donc faire appel au DOS ou au BIOS. Le DOS ne fait que transmettre au BIOS. Il faut donc utiliser le BIOS. L'interruption 14h est là pour cela, et le problème de vitesse est ainsi résolu : 119 000 bauds ne vous feront pas peur. Mais si vous souhaitez d'autres fonctions que celles proposées par le BIOS, par exemple gérer la ligne par interruption, ou utiliser une fonction de test, analogue à KEYPRESSED, il faut encore descendre d'un cran, et programmer directement les registres de la carte série.

Le programmeur PASCAL est donc bien obligé dans certaines circonstances de descendre de sa stratosphère et de se plonger dans les profondeurs du système.

Le principal obstacle à ce niveau est alors la documentation : comment fonctionne tel périphérique, dans quel registre placer telle information, où se trouve telle table...?

Le présent livre offre alors une mine d'informations pour arriver à comprendre, utiliser, modifier la gestion des mémoires du PC en PASCAL. Ceux qui aiment savoir «comment ça marche» y trouveront de nombreuses informations. Mais surtout, il expose concrètement comment utiliser ces informations dans des programmes PASCAL. Il vous permettra ainsi de réaliser de nombreux utilitaires pour gérer dans des programmes PASCAL la mémoire centrale et les disquettes ou disques durs.

Sommaire

Présentation

Ces quelques pages ont pour but de présenter le livre et de faire le point sur ce qu'il est nécessaire de connaître avant d'aborder la programmation système.

Au sujet de ce livre

Programmation système a été écrit pour combler un manque : s'il existe d'innombrables ouvrages sur le DOS et le BIOS, peu prennent la peine d'expliquer comment le système travaille de l'intérieur et de quelle façon le programmeur peut en tirer profit. Et aucun, à l'exception de quelques ouvrages américains, ne présente de programme complet faisant appel aux fonctions les plus intéressantes du DOS ou du BIOS.

Or l'écriture de certains utilitaires serait impossible sans passer par ces fonctions : imagine-t-on d'écrire un programme de récupération de disque sans connaître la FAT, le secteur de boot et le format des entrées-fichiers ? Ce livre présente de telles structures, mais surtout il explique comment les mettre en œuvre par l'intermédiaire d'une quarantaine de programmes d'exemples listés, commentés et fournis sur la disquette d'accompagnement.

Outre les structures de données "classiques" du DOS (comme la FAT et le secteur de boot), nous examinerons également ses structures secrètes (nœud d'informations, MCB, *System file table*, etc.) et celles du BIOS.

Chacune de ces structures relevant soit de la mémoire vive, soit des disques, soit des fichiers, nous nous sommes attachés à étudier ces trois thèmes principaux. Comme il a bien fallu s'arrêter à un moment donné, certains sujets spécifiques, comme les *device drivers* ou la mémoire étendue (EMS et XMS), n'ont pas été approfondis.

De quoi traite ce livre

Après avoir lu cet ouvrage, vous devriez être en mesure de comprendre comment fonctionnent le DOS, le BIOS et la plupart des utilitaires du marché (comme les programmes de *PcShell* ou les *Norton utilities*). Vous devriez même être suffisamment entraîné aux techniques auxquelles ils font appel pour programmer les mêmes utilitaires.

Vous aurez vu en effet :

1. les concepts de base de la programmation système ;
2. les données du BIOS en RAM ;
3. la RAM gérée par le DOS ;
4. les disques au niveau physique ;
5. le plan d'un disque au niveau logique ;
6. les structures DOS de bas niveau de gestion des disques au niveau logique (FAT, secteur de boot) ;
7. les structures DOS de haut niveau de gestion des disques au niveau logique (répertoires, etc.) ;
8. les fichiers de données ;
9. les fichiers `.EXE`.

Les concepts de base — Le *chapitre 1* s'intéresse à la définition de ce qu'il est convenu d'appeler "programmation système". Il introduit à la vision en couches de l'ordinateur et explique le rôle et le fonctionnement des interruptions.

Les données du BIOS en RAM — Le *chapitre 2* fait l'inventaire des diverses données que le BIOS met à la disposition des programmeurs pour leur indiquer l'état du système. Un programme, non listé mais présent sur la disquette, permet de visualiser ces données sur chaque PC 100% compatible IBM.

La RAM gérée par le DOS — Le *chapitre 3* s'intéresse au fonctionnement interne de la mémoire système. Il examine chacune des principales structures de données mises en place par le DOS : de la table des vecteurs d'interruption à la liste des buffers disque en passant par les blocs de paramètres disque et les device

drivers internes au DOS, toutes les structures cachées du système d'exploitation sont révélées, leur format détaillé, leur fonctionnement expliqué et illustré par un court programme.

Ensuite de quoi, ce sont les mécanismes de gestion de la mémoire utilisateurs qui sont examinés. La fonction EXEC, les fonctions d'attribution de la mémoire, les MCB et les PSP prennent place dans cette seconde partie, qui se termine avec un programme permettant de dumper et de modifier le contenu de la mémoire du PC.

Les disques au niveau physique — Le *chapitre 4*, premier de la partie consacrée aux disques, fait l'inventaire des diverses fonctions BIOS de gestion des disques et disquettes. Il contient un programme de formatage physique de disquette.

Les disques au niveau logique : le plan d'un disque — Le *chapitre 5* fait le point sur la terminologie employée en matière d'organisation logique des disques et disquettes et livre un programme permettant de dumper et de modifier le contenu des secteurs logiques d'un disque.

Les disques au niveau logique : structures DOS de bas niveau — Le *chapitre 6* explique le rôle et le fonctionnement de chacune des trois structures primordiales d'un disque :

1. le secteur de boot (avec un désassemblage commenté du programme de boot d'une disquette) ;
2. la FAT au format 12 et 16 bits (avec un programme listant les valeurs de la FAT pour un fichier donné) ;
3. et la table de partition d'un disque dur (avec un désassemblage du secteur de partition et un programme affichant la table des partitions).

Les disques au niveau logique : structures DOS de haut niveau — Le *chapitre 7* s'intéresse au format et au fonctionnement des structures de données que sont les entrées-fichiers, le répertoire racine, les sous-répertoires et la zone des données. Il livre, entre autres, un programme de récupération d'un fichier effacé.

Les fichiers de données — Le *chapitre 8* fait le point sur la gestion de fichiers par handles. Il passe en revue les différentes fonctions concernées de l'Int 21h. Il expose également comment le DOS gère les handles et sur quelle structure de données ceux-ci pointent. La *File handle table* et la *System file table*, leur format et leurs principes de fonctionnement sont détaillés. Des programmes accédant à ces structures de données et les interprétant sont listés. Enfin, les principes de la redirection et des filtres sont expliqués et donnent lieu à l'écriture d'un programme recherchant une chaîne dans un fichier texte ou binaire.

Les fichiers .EXE — Le *chapitre 9* passe en revue toutes les phases de la vie d'un fichier .EXE, de sa création à son exécution. Au fur et à mesure, on apprend comment fonctionne l'en-tête d'un fichier .EXE, comment il permet de retrouver les procédures far à l'intérieur du fichier (très utile lors d'un désassemblage), comment s'effectue le relogement des adresses et quelles sont les étapes par lesquelles passe

un fichier `.EXE` avant d'être chargé en mémoire. Divers programmes interprétant l'en-tête d'un fichier `.EXE`, affichant les adresses à reloger, modifiant les valeurs des champs `MinAlloc` et `MaxAlloc`, sont listés.

Comment lire ce livre

Il est bien connu que les programmeurs ne lisent jamais un livre du début à la fin, mais s'en servent systématiquement comme d'un ouvrage de référence. Les auteurs d'ouvrages informatiques sont malgré tout obligés de commencer quelque part et de suivre un cheminement logique.

C'est ce qui a été fait ici : on peut donc lire ce livre en commençant au premier chapitre, en continuant par le second et en terminant par la dernière annexe (juste avant l'index).

On peut aussi le lire thème par thème, ce qui serait la meilleure démarche. En effet, les thèmes – et, à l'intérieur de chaque thème, les chapitres – se succèdent dans un ordre qui commence toujours par le général pour aboutir au particulier. Chaque grand sujet (mémoire vive, disques et disquettes, fichiers) formant un tout, il serait préférable de commencer un thème par son premier chapitre et de le terminer par le dernier. En revanche, on peut tout à fait commencer par le dernier thème et terminer par le premier, même si ce n'est pas de cette façon que la lecture de ce livre a été prévue.

Les annexes

Les annexes qui se trouvent en fin d'ouvrage sont de même importance que les chapitres. Elles n'ont été rejetées dans les dernières pages que pour des raisons d'équilibrage des chapitres et de communauté d'intérêt des informations qu'elles contiennent. Le lecteur perspicace remarquera l'absence d'une table ASCII : cet oubli est volontaire. Il a permis de regagner deux pages qui ont été employées à des fins plus utiles.

Les programmes

Les programmes de ce livre sont pour la plupart écrits en *Turbo Pascal version 5.5*, et pour quelques uns en *Turbo Assembler version 1.0*. Tous sont compatibles avec *Turbo Pascal version 4.0, 5.0* et *6.0*, ainsi qu'avec *QuickPASCAL*. Les programmes Assembleur, pour leur part, sont compatibles avec *Turbo Assembleur version 2.0*, et

au prix de quelques modifications de détail avec *Microsoft Macro-Assembler version 4.0, 5.0* et *5.1*.

Ces programmes sont de deux sortes :

1. des programmes visualisant les structures internes du BIOS et du DOS ;
2. des programmes mettant en œuvre ces structures dans un but utilitaire.

Les premiers sont à faire fonctionner absolument si l'on veut comprendre à quoi servent les structures mises à jour, comment y accéder et comment les interpréter.

Les seconds sont à lire, à taper, à exécuter selon certaines restrictions données dans le corps du chapitre et à modifier. Si vous vous contentiez de charger ces programmes et de les exécuter, il y aurait de grandes chances pour que les techniques de programmation auxquelles ils font appel restent longtemps du domaine de l'inconnu. C'est aussi pourquoi nous suggérons que vous modifiiez ces programmes pour y ajouter des fonctionnalités, lever certaines de leurs limitations et en faire de véritables utilitaires que vous pourrez utiliser tous les jours en lieu et place des ordres du DOS ou des *Norton Utilities*.

A quels lecteurs s'adresse ce livre

Programmation système n'a pas été écrit pour des débutants en programmation. Ses lecteurs sont supposés avoir acquis une bonne expérience du langage Pascal et du DOS. Ni les fichiers .BAT, ni les Tpu n'ont de secrets pour eux. En outre, ils doivent avoir déjà eu la curiosité de lire des programmes en Assembleur. Le mieux serait qu'ils aient déjà écrit un ou deux petits programmes affichant une chaîne de caractères à l'écran dans ce langage, et qu'ils aient lu les quatre premiers chapitres du *Manuel de l'utilisateur du Turbo Assembler*.

Enfin, deux ouvrages sont essentiels à la bonne utilisation de celui-ci : ce sont la seconde édition de *Advanced MS-DOS Programming*, par Ray Duncan, et *The New Peter Norton Programmer's Guide to the IBM PC & PS/2*, par Peter Norton et Richard Wilton. S'il vous en manque un, achetez-le. De toutes les façons, les deux sont indispensables si l'on souhaite programmer système. Un simple coup d'œil à la bibliographie (*Annexe 3*) vous convaincra vite qu'il ne s'agit pas d'un luxe, eu égard aux nombreux autres qui s'y trouvent, et que vous lirez sûrement un jour si vous persistez dans ce type de programmation.

Chapitre 1

Concepts de base

Mots-clefs

BIOS	*Basic Input/Output System,* le logiciel de plus bas niveau, logé en mémoire morte (ROM), qui sert d'interface entre le matériel et le DOS.
Couches (vue en)	Méthode utilisée pour distinguer les différents éléments qui composent un ordinateur en fonctionnement. Elle a l'intérêt d'être simple à comprendre et de présenter une organisation hiérarchique.
DOS	*Disk Operating System,* le système d'exploitation le plus courant sur PC. Il est chargé en mémoire vive (RAM) et sert d'interface entre les logiciels d'application et le BIOS.
Interruption	Mécanisme qui permet aux différents éléments (matériel, BIOS, DOS, logiciels d'application) de communiquer entre eux et de passer le contrôle à un élément d'un autre niveau.
Logiciel	Lorsque ce terme est synonyme d'application ou logiciel d'application, c'est un programme exécutable quelconque qui se situe au dessus du DOS. Sinon, il signifie seulement "programme", par opposition aux données et au matériel.
Matériel	Les éléments mécaniques et électroniques du PC. Le BIOS est un logiciel chargé de gérer ces éléments.
RAM	*Random Access Memory,* aussi appelée "mémoire vive", c'est la partie de la mémoire qui contient le DOS, les logiciels et les données qu'ils manipulent. Son contenu (modifiable) s'efface à la moindre coupure de courant.
ROM	*Read Only Memory,* aussi appelée "mémoire morte", c'est la partie de la mémoire qui contient le BIOS. On ne peut pas la modifier et son contenu ne s'efface pas lorsqu'on coupe le courant (heureusement, sinon comment lancer le PC ?).
Structures de données	Elles permettent de regrouper les données et de les organiser. Le PSP est une structure de données mise en place par le DOS.

Ce premier chapitre va vous présenter quelques concepts-clefs liés à la programmation système. Après avoir vu de quels éléments est composé le système, nous serons à même de définir la programmation système telle que nous l'entendons et de cerner ses qualités et ses défauts.

Qu'est-ce que le système ?

Il y a plusieurs façons de définir le système. En procédant par éliminations, on peut dire qu'il relève du logiciel et pas du matériel. C'est insuffisant, pour la simple raison que Word, par exemple, est un logiciel, mais ne fait pas partie du système. On doit donc affiner notre vision des choses : nous dirons alors que le système ne relève ni du matériel ni du logiciel d'application.

Cela permet d'éliminer tous les logiciels du commerce, ainsi que les programmes les plus couramment écrits. Mais cela ne dit pas exactement ce qu'est le système. Sans compter que, dans certains cas, cela prête à confusion : un disque RAM, géré par un pilote de périphérique comme `RAMDRIVE.SYS`, fait-il partie du système ? Non, et pourtant un pilote de périphérique n'est à proprement parler ni un logiciel d'application, ni un programme courant. Et un disque RAM n'est pas du matériel. Alors qu'est-ce que le système ? En fait, la méthode par éliminations n'est pas la meilleure si l'on souhaite éviter les confusions.

La vision en couches

Une façon simple de comprendre comment fonctionne un micro-ordinateur – et quels sont les éléments qui le composent – est d'adopter ce que Andrew Tanenbaum [1] appelle la "vision en couches". Sans aller aussi loin que lui dans ce sens (il distingue six couches), nous allons en présenter les grands principes.

---

(1) Andrew Tanenbaum est l'une des grandes figures de l'informatique, qu'il enseigne à l'Université libre d'Amsterdam. Il est l'auteur de trois livres principaux (*Structured Computer Organisation, Operating Systems - Design and Implementation,* et *Computer Networks*) parus chez Prentice Hall. Ses ouvrages font figure de bible dans leur domaine. Il a en outre réalisé un compilateur portable d'un matériel à l'autre, et un système d'exploitation (Minix) disponible sur PC et Atari.

Un ordinateur (gros système, mini ou micro-ordinateur) comporte plusieurs couches qui se superposent les unes aux autres (voir *figure 1.1*). Ces couches sont :

1. le matériel,
2. le BIOS,
3. le système d'exploitation (DOS, UNIX, OS/2, etc.),
4. les logiciels.

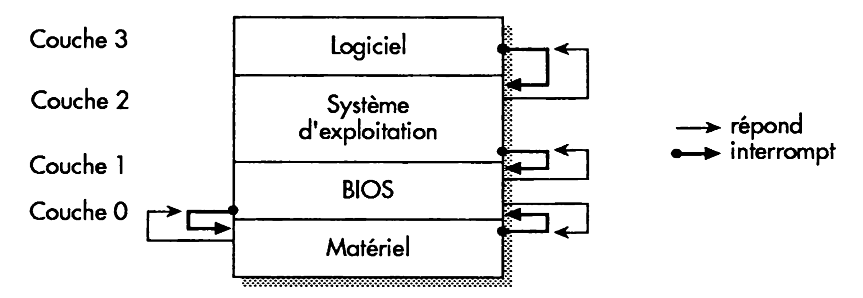


Figure 1.1
Les couches d'un ordinateur.

Si nous regardons la *figure 1.1*, nous pouvons émettre plusieurs remarques :

1. le système n'y apparaît pas ;
2. les couches vont de bas en haut : la plus basse, numérotée 0, est la couche matérielle. La plus haute, numérotée 3, concerne les logiciels ;
3. le BIOS est proche du matériel (couche basse), le système d'exploitation (pour nous, le DOS) se situe entre le BIOS et les logiciels.

Fort logiquement, nous en déduisons :

1. le BIOS gère les aspects matériels du fonctionnement du PC ;
2. le DOS sert de passerelle (ou encore, d'interface) entre les logiciels et le BIOS ;
3. plus nous nous rapprochons des logiciels et plus nous nous éloignons du matériel, ce qui signifie que nous nous assurons une certaine indépendance de fonctionnement du logiciel quant à l'architecture matérielle du PC et donc quant à d'éventuelles incompatibilités.

Deux couches, un système

Dès lors, nous pouvons définir le système de la façon suivante : c'est la réunion du BIOS et du système d'exploitation. Le système est proche de la machine et dépend de son architecture matérielle. En fait, c'est un logiciel qui gère le matériel (BIOS), qui offre au programmeur l'organisation des données, une certaine indépendance

matérielle, et la gestion automatique de l'exécution des applications en mémoire (DOS).

Le système est donc composé de deux éléments principaux qui appartiennent à une couche différente. Chacun de ces éléments doit être en mesure de communiquer avec la couche inférieure et la couche supérieure. Si ce n'était pas le cas, il serait impossible à un programme écrit par l'utilisateur de faire une action aussi simple qu'effacer l'écran : en effet, celui-ci relève du matériel et du programme utilisateur du logiciel. Si la communication se rompait quelque part entre la couche logicielle et la couche matérielle, l'ordre ne pourrait pas être transmis à l'écran, lequel ne pourrait pas s'effacer. Or nous avons vu que le logiciel devait lui-même être indépendant du matériel. Comment concilier ces deux obligations : indépendance vis-à-vis du matériel et communication inter-couches ?

Communication inter-couches

On a résolu la contradiction en offrant au logiciel la possibilité de communiquer un ordre au DOS, au DOS celle de communiquer un ordre au BIOS et au BIOS celle de communiquer un ordre au matériel. Est-ce le même ordre qui est répété trois fois ? Pas exactement.

Le DOS permet à un programme de lui demander d'afficher une chaîne de caractères à l'écran. Cette chaîne de caractères peut être une séquence de codes ANSI. Lorsque le DOS reçoit cette chaîne de caractères, il doit l'interpréter pour lui donner son véritable sens [2]. Une fois qu'il l'a reconnue comme étant une demande d'effacement de l'écran, il donne les ordres qui s'imposent au BIOS.

1. Faire défiler la fenêtre entière vers le haut,
2. positionner le curseur en `0,0` page `0`,
3. afficher la page `0`.

Le BIOS, lui, reçoit chacun de ces trois ordres les uns après les autres et les traduit au matériel. Ainsi, pour positionner le curseur, il lui donne quatre ordres, qui supposent tous que différents tests aient été effectués avec succès.

On le voit, plus on se rapproche du matériel et plus on est obligé d'entrer dans les détails. Encore faut-il avoir présent à l'esprit que nous avons pris un exemple simple, et que nous n'avons pas parlé des nombreux tests qui sont opérés en cours de route (voir la *figure 1.2*).

---

(1) Ici, nous schématisons. En fait, lorsque COMMAND.COM reconnaît l'ordre CLS, il peut soit exécuter directement les 3 ordres BIOS, soit (si ANSI.SYS est chargé) envoyer la chaîne de caractères de contrôle au pilote d'écran.

1. Ordre utilisateur passé en ligne de commande

```
C:\> CLS <┘                         ; Ordre utilisateur
```

2. Ordres passés par `COMMAND.COM` au noyau du DOS

```
.
EffacerEcran   Db 1Bh'[2J$'   ; Codes ANSI d'effacement
.                             ; Données COMMAND.COM
.
.
Mov    Ah, 09h
Mov    Dx, SEG EffacerEcran
Mov    Ds, Dx
Mov    Dx, OFFSET EffacerEcran
Int    21h
```

3. Ordres passés par le noyau du DOS au BIOS

```
                          ; 3.1 Défiler vers le haut
Mov    Ah, 06h            ; tout l'écran
Mov    Al, 00h
Mov    Bh, 07h
Mov    Ch, 00h
Mov    Cl, 00h
Mov    Dh, 18h
Mov    Dl, 4Fh
Int    10h

Mov    Ah, 02h           ; 3.2 Positionner le curseur
Mov    Bh, 00h           ; en 0,0 page 0
Mov    Dh, 00h
Mov    Dl, 00h
Int    10h

Mov    Ah, 05h           ; 3.3 Afficher la page 0 à l'écran
Mov    Al, 00h
Int    10h
```

4. Ordres passés par le BIOS au contrôleur de l'écran pour positionner le curseur en `Dx` Page `Bh`

```
Mov    Al, Bh
CbW
Shl    Ax, 1
XChg   Ax, Si
Mov    [Si+OFFSET PositionCurseur], Dx
Cmp    PageActive, Bh
Jne    FinAppel
```

```
Mov     Ax, Dx
Push    Bx
Mov     Bx, Ax
Mov     Al, Ah
Mul     Byte Ptr NbColEcran
Mov     Bh, 00h
Add     Ax, Bx
Shl     Ax, 1
Pop     Bx
Mov     Cx, Ax
Add     Cx, AdresseDeBasePageActive
Shr     Cx, 1
Mov     Ah, 14
Mov     Dx, AdresseControleurEcran
Mov     Al, Ah
Out     Dx, Al          ; Premier ordre au contrôleur
Inc     Dx
Mov     Al, Ch
Out     Dx, Al          ; Second ordre
Dec     Dx
Mov     Al, Ah
Inc     Al
Out     Dx, Al          ; Troisième ordre
Inc     Dx
Mov     Al, Cl
Out     Dx, Al          ; Quatrième ordre
FinAppel :
Jmp     Fin
```

Figure 1.2
Succession des événements, de l'ordre utilisateur aux actions du matériel.

Il faut bien avoir présent à l'esprit que si l'on ne passait pas par le DOS, puis le BIOS, pour aboutir au matériel, l'utilisateur d'un micro-ordinateur serait obligé de passer ses ordres lui-même au matériel, et cela en binaire. C'est d'ailleurs ce qui se faisait aux temps préhistoriques de la micro-informatique. Cela n'a pas duré.

Maintenant que la nécessité d'établir plusieurs couches et celle de les faire communiquer entre elles est établie, il reste à identifier le moyen de communication inter-couches. Ce moyen est mis en œuvre sur le PC par le mécanisme des interruptions.

Les interruptions

Une interruption peut être de deux sortes : matérielle ou logicielle. Les interruptions matérielles sont des signaux qu'adresse le matériel au logiciel pour le prévenir qu'il est prêt à travailler, ou au contraire qu'il est occupé. Les interruptions logicielles servent de passerelles entre les applications, le DOS et le BIOS.

Mais quelle que soit leur origine, toutes fonctionnent de la même façon. Elles sont repérées par un numéro d'ordre et acceptent en entrées les paramètres qui leur sont passés dans les registres du PC. Lorsqu'une application exécute une interruption (comme COMMAND.COM, quand il affiche une chaîne de caractères, qui se trouve être un ordre d'effacement de l'écran), elle s'arrête de travailler pendant un temps indéterminé, et le niveau inférieur prend le relai.

En effet, à chaque fonction d'une interruption correspond un sous-programme qui se trouve *à une couche inférieure* à celle qui a appelé cette fonction. Ainsi, le sous-programme de la fonction d'affichage d'une chaîne de caractères demandant l'effacement de l'écran est la suite de trois interruptions BIOS qu'exécute le noyau du DOS. Et à chacune de ces interruptions BIOS correspond un sous-programme tel que celui qui positionne le curseur. Lorsque le matériel a terminé, il rend la main et le contrôle revient au BIOS. Lorsque le BIOS a terminé, il retourne le contrôle au DOS, qui finit son travail et autorise de nouveau l'utilisateur (ici, COMMAND.COM) à continuer. Il s'agit donc d'un mécanisme en cascade (voir page ci-contre).

Naturellement, chaque fonction d'une interruption doit procéder à ses propres tests et peut donc échouer. Elle se termine alors en renvoyant un code d'erreur dans un registre spécifique, qui peut (et doit) être testé par l'appelant. Pour effectuer ces tests ou pour servir l'interruption, il faut souvent disposer de données qui ne peuvent que transiter par les registres et non y être mémorisées. Ces données sont regroupées quelque part en mémoire (les données du BIOS en 0040h:0000h jusqu'à 0040h:0100h, les données du DOS dans divers endroits) selon une structure propre. Le PSP, par exemple, est l'une de ces structures. Il mémorise certaines informations dont peuvent avoir besoin le programme utilisateur et le DOS lui-même et se trouve chargé en mémoire 256 octets avant le programme utilisateur.

Résumons-nous : nous savons comment et pourquoi l'ordinateur se partage en couches. Nous savons également comment ces couches communiquent et ce qu'il se passe lorsqu'une interruption (une demande de communication) se produit. Nous savons enfin que le système a besoin de mémoriser certaines données pour répondre aux sollicitations de l'utilisateur. Bref, les grandes lignes du fonctionnement du système nous sont connues. Mais qu'est-ce que la programmation système ?

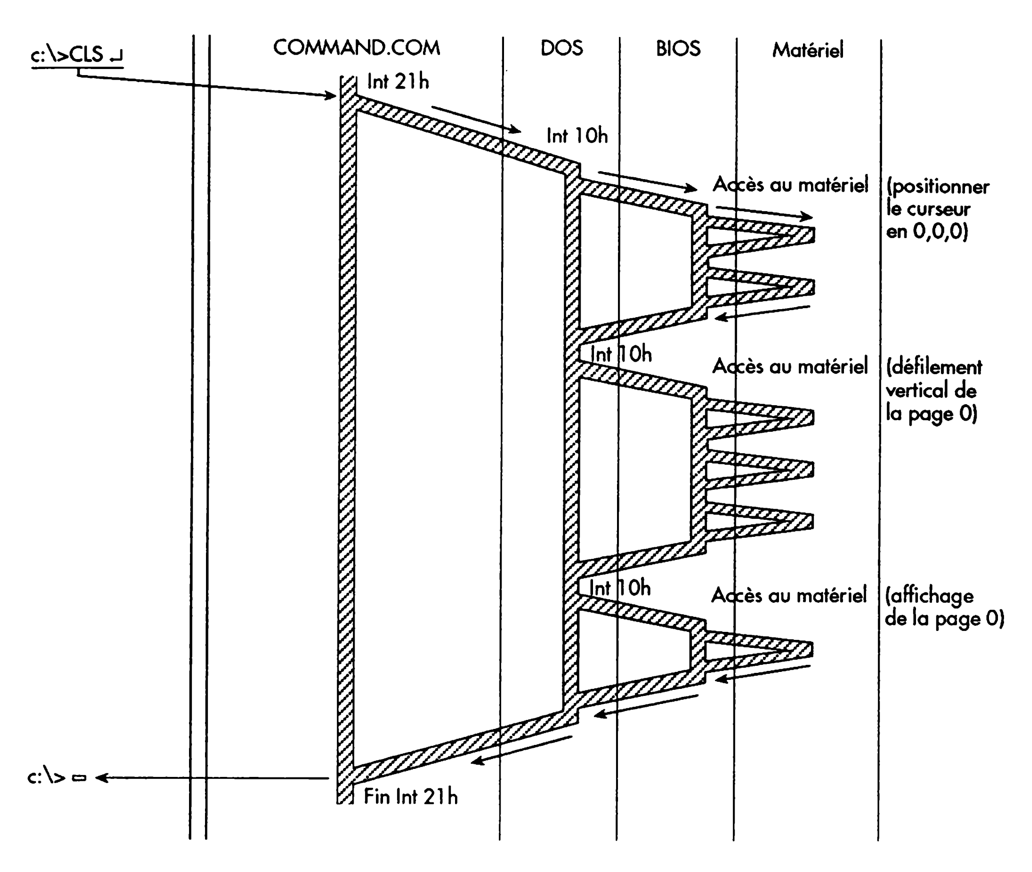


Figure 1.3
Le mécanisme des interruptions.

La programmation système

Le DOS et le BIOS, éléments principaux du système, ont été programmés : ce sont des logiciels écrits en Assembleur et en C. Même s'ils se tiennent entre la couche machine et la couche logicielle, il est possible d'écrire des programmes qui travaillent à leur niveau. Ainsi, la plupart des applications graphiques et des device drivers (pilotes de périphériques) sont programmés au niveau du BIOS et/ou du matériel.

On peut considérer qu'il s'agit de programmation système, parce qu'il s'agit de programmes qui ne disposent pas des ressources offertes par le DOS aux programmes d'un niveau supérieur. En revanche ils disposent des mêmes ressources

et accèdent aux mêmes données que le DOS. Ce sont donc des programmes plus longs à écrire que les programmes de la couche logicielle (mais pas vraiment plus compliqués).

Comme nous allons surtout programmer en Turbo Pascal, nous nous trouverons dans une situation intermédiaire : la plupart des services du DOS nous sont ouverts, ainsi que ses structures de données. En bref, nous avons réuni le maximum d'avantages et le minimum d'inconvénients des deux types de programmation. Programmer système a donc deux significations principales :

1. programmer au niveau du système,
2. utiliser les données du système.

Les données du système

Le BIOS et le DOS disposent de certaines données dont ils ont besoin pour mener leur travail à bien. Alors que le BIOS les stocke dans un endroit fixe de la mémoire, le DOS les enregistre en plusieurs endroits différents, ce qui les rend parfois plus difficiles à trouver.

Par exemple, le BIOS conserve en `0040h:0013h` la quantité de mémoire vive (RAM) du PC. Il propose également une interruption (Int `12h`) renvoyant la quantité de RAM du PC dans le registre `AX`. Lorsque le BIOS reçoit une demande de servir l'Int `12h`, il lit le Mot (`Word`) qui se trouve en `0040h:0013h`, le copie dans le registre `AX` et rend la main à l'appelant. Il y a donc plusieurs manières de connaître la quantité de RAM dont dispose un PC. On peut en effet :

1. lire le mot qui se trouve en `0040h:0013h`,
2. exécuter une Int `12h`.

Chacune de ces possibilités pouvant être programmée en Pascal ou en Assembleur, nous allons commencer par l'Assembleur.

Listing 1.4

Afficher la taille de la RAM en lisant les données BIOS. ❶

```
; MemBios.Asm : Lit le Mot à l'Adresse 0040h:0013h,        ;
;               et affiche la taille de la RAM
;
;                                                           ;
.Model Small
.Data
 Chaine    Db  ' Ko de RAM ',0dh, 0ah, '$'
.Stack 100h
```

Afficher la taille de la RAM en lisant les données BIOS (suite). ❷

```
.Code

Debut:                              ; Debut du programme
      Mov    Ax,@Data
      Mov    Ds,Ax                  ; Données en DS
      Mov    Dx,0040h               ; Données BIOS en 0040h
      Mov    Es,Dx
      Mov    Ax,Es:[0013h]          ; Mot en 0040h:0013h
      Mov    Cx,0004h
Bcle:                               ; Convertir Hexa en décimal
      Xor    Dx,Dx
      Mov    Si,0ah                 ; Diviser DX:AX par 10
      Div    Si
      Push   Dx                     ; Sauver le Reste
      Loop   Bcle                   ; 4 fois
      Mov    Cx,0004h
Affiche:
      Pop    Dx                     ; Afficher Taille
      Add    Dl,30h                 ; Convertir chiffre en ASCII
      Mov    Ah,02h                 ; Afficher 1 caractère
      Int    21h
      Loop   Affiche                ; 4 fois
      Mov    Ah,09h                 ; Afficher une chaîne
      Mov    Dx,SEG Chaine
      Mov    Ds,Dx
      Mov    Dx,OFFSET Chaine ; Chaîne en DS:DX
      Int    21h
      Mov    Ax,4c00h               ; Terminer, Erreur 0
      Int    21h
      End    Debut                  ; Commencer en Debut
```

Si l'on utilise l'Int 12h, toujours en Assembleur, cela donne le listing suivant.

Listing 1.5

Afficher la taille de la RAM obtenue par l'Int 12h. ❶

```
; MemInt.Asm : Exécute une Int 12h pour connaître ;
;              la taille de la RAM et l'affiche   ;
;                                                 ;

.Model Small
.Data
  Chaine          Db ' Ko de RAM ',0Dh,0Ah,'$'
.Stack 100h
.Code
```

➥

Afficher la taille de la RAM obtenue par l'Int 12h (suite). ❷

```
Debut:
    Mov  Ax, @Data
    Mov  Ds, Ax
    Int  12h              ; Exécute l'Int 12 du BIOS
    Push Ax               ; Ax:=Taille de la RAM
    Mov  Cx, 0004h
Bcle:
    Xor  Dx, Dx
    Mov  Si, 0Ah          ; Divisions par 10
    Div  Si
    Push Dx               ; Dx:= (Dx:Ax MOD Si)
    Loop Bcle             ; Boucler tant que Cx > 0
    Mov  Cx, 0004h        ; Afficher quatre chiffres
Affiche:
    Pop  Dx
    Add  Dl, 30h          ; ASCII
    Mov  Ah, 02h          ; Affichage
    Int  21h
    Loop Affiche

    Mov  Ah, 09h          ; Afficher une Chaîne
    Mov  Dx, SEG Chaine
    Mov  Ds, Dx
    Mov  Dx, OFFSET Chaine
    Int  21h              ; Affichage

    Mov  Ax, 4C00h        ; Fin de Programme, ErrorLevel 0
    Int  21h
    End  Debut
```

Traduit en Pascal, le *listing 1.4* donnerait cela :

Listing 1.6

Lit la variable BIOS en 0040h:0013h et l'affiche.

```
PROGRAM TailleRam;        { MemBios.Pas }
VAR  Memoire : Word;

BEGIN
 { MemW est un tableau de Mots, qui représente la mémoire }
 { Il s'indice en spécifiant un Segment et un déplacement }
 {         Ici, les adresses sont en hexadécimal          }
  Memoire:=MemW[$0040:$0013];
 { Il n'y a pas de conversion à faire : Pascal affiche    }
 {      les nombres sous la forme décimale par défaut     }
  WriteLn('Vous avez ', Memoire, ' Ko de RAM');
END.
```

Enfin, la traduction en Pascal du *listing 1.5* donne le programme du listing suivant.

Listing 1.7

Lit la taille de la RAM par l'Int 12h du BIOS et l'affiche.

```
PROGRAM TailleRam           { MemInt.Pas }

USES Dos;                   { Pour les registres et les Ints }

FUNCTION Ram : Word;        { Ax est un Word }
VAR Regs : Registers;
BEGIN
  Intr($12, Regs);          { Pas de paramètres à passer }
  Ram := Regs.Ax;           { Renvoie Ax }
END;

BEGIN
  WriteLn('Vous avez ', Ram, ' Ko de Ram ');
END.
```

C'est en analysant ces quatre programmes que nous pourrons cerner les qualités et les défauts de la programmation système.

Qualités et défauts de la programmation système

Dans les *listing 1.4* et *1.6*, nous évitons de passer par l'interruption `12h` du BIOS. Le principal intérêt de cette méthode est sa rapidité : en effet, le programme n'est pas interrompu lors de la recherche de la taille mémoire du PC. Au lieu de passer par une interruption, il fait une simple lecture de la mémoire. Dans le cas contraire, il se serait arrêté le temps pour le BIOS de :

1. reconnaître le numéro de l'interruption,
2. sauter à la procédure chargée de la servir,
3. faire une lecture mémoire,
4. sauver le résultat en AX,
5. rendre la main au programme appelant.

D'un autre côté, tous les BIOS ne sont pas compatibles avec celui d'IBM (ainsi, celui d'HP). Ce qui signifie que les renseignements d'un BIOS donné ne se situent pas forcément au même endroit que ceux d'un autre BIOS. Les programmes *1.4* et *1.6*

peuvent donc ne pas fonctionner sur toutes les machines. Si l'on passe par les interruptions, ce risque est quasiment annulé mais on ralentit le programme.

Autant ralentir un programme écrit en Pascal peut être gênant, autant ralentir un programme écrit en Assembleur ne porte pas à conséquence dans la plupart des cas. Il vaut donc mieux utiliser les données BIOS lorsqu'on programme en Pascal et les interruptions si l'on écrit en Assembleur. Naturellement, cela dépend de la donnée à lire – certaines se trouvent au même endroit dans tous les BIOS (IBM, Award, Phœnix, HP), d'autres non – et de l'importance que revêt la portabilité du programme écrit.

En outre, il faut également tenir compte du ralentissement relatif provoqué par les interruptions. Dans le *listing 1.7*, ajouter une interruption ou en supprimer une revient presque au même : ce sont l'initialisation, la conversion d'un nombre en caractères et l'affichage de la chaîne qui prennent le plus de temps. Ajouter une interruption n'est donc pas grave dans ce cas-ci. Mais la taille de la RAM fait justement partie des données qui se trouvent au même endroit dans tous les BIOS...

Que faire ? Si le programme était long, chaque minute gagnée serait importante, et l'on aurait tendance à passer au maximum par les données du BIOS. Dans ce cas-ci, où le programme fait une taille totale de trois lignes, mieux vaut priviligier la compatibilité que la rapidité et utiliser les interruptions.

D'une manière générale, ce dont il faut absolument se souvenir, c'est que la programmation système a l'avantage de la rapidité et de l'économie d'espace mémoire et l'inconvénient de l'incompatibilité. La structure interne du DOS est bouleversée d'une version à l'autre et les adresses de ses données changent d'un constructeur à l'autre. Ni vos programmes systèmes, ni ceux du livre ne seront donc *entièrement* compatibles. Ceux du livre ont été testés avec un DOS Microsoft et un DOS IBM 3.3 et obéissent aux particularités des deux standards.

Conclusion

Les concepts de base de la vision en couches, du système et de la programmation système ont été vus. Si vous n'étiez pas au point en ce qui concerne l'architecture du 8086/8088, le mieux serait de lire le chapitre 3 du *Guide de l'utilisateur du Turbo Assembler*, qui contient tout ce qu'il faut savoir sur les registres, la segmentation mémoire et le reste.

Chapitre 2

Données du BIOS en RAM

Le BIOS est contenu dans la ROM. Mais il fait souvent appel à des données pouvant également servir au système d'exploitation ou aux programmes utilisateurs : ces données sont stockées en RAM, de l'adresse 0040h:0000h à l'adresse 0040h:0100h. Nous en faisons ici l'inventaire.

Données POST (*Power-On Self-Test*)

Mieux vaut ne pas modifier ces données : elles sont vitales pour le BIOS puisqu'elles lui indiquent comment s'est déroulé le lancement du système.

Description	*Adresse*		*Type*
Test fabricant (AT)	0040h:0012h		octet
Test fabricant (AT)	0040h:0015h		octet
Pointeur sur routine d'initialisation	0040h:0067h		2 mots
Une interruption a eu lieu	0040h:006Bh		octet
Drapeau de reset du système (AT)	0040h:0072h		octet
	= 1234h	pas de test mémoire (boot *à chaud*)	
	= 4321h	préserve la mémoire	
	= 64h	boot *à froid*	
Réservé (AT)	0040h:00B6h		3 octets

Tableau 2.1
Données POST.

Un moyen connu de redémarrer le système au clavier sans passer par la combinaison de touches Alt-Ctrl-Del, est d'initialiser le drapeau de reset ([0040h:0072h]) à 1234h et d'effectuer un Jump en FFFFh:0000h. Programmé en Assembleur et compilé par DEBUG, cela tient en seize octets et se présente ainsi :

```
;Reset.Asm en 16 octets, utilise la variable du BIOS
;en [0040h:0072h], pour rebooter le PC

a                               ; Pour assembler
mov ax,0040                     ; DS := 0040
mov ds,ax
mov word ptr [0072],1234        ; DS:[0072] := 1234h
jmp ffff:0000                   ; Saut en début de ROM
```

```
                                 ; Ligne blanche nécessaire
n reset.com                      ; Nommer le fichier
r cx                             ; Changer CX
0010                             ; 16 octets (10h)
w                                ; Ecrire Reset.Com
                                 ; Ligne blanche nécessaire
q                                ; Quitter DEBUG
```

Après avoir tapé ceci sous l'éditeur du Turbo Pascal (sauf les commentaires), il vous suffit de passer au DOS la commande :

```
C:\>Debug < Reset.Asm
```

pour disposer du fichier `Reset.Com`.

Données concernant l'équipement

Ces données renseignent le BIOS et le système sur la configuration du PC :

Description	*Adresse*	*Type*
Adresse des ports RS/232	`0040h:0000h`	4 mots
Adresse des ports imprimantes	`0040h:0008h`	4 mots
Périphériques installés	`0040h:0010h`	1 mot (voir description)
Drapeau d'initialisation (AT)	`0040h:0012h`	1 octet
Taille mémoire	`0040h:0013h`	1 mot
Test fabricant POST (AT)	`0040h:0015h`	1 octet
Codes d'erreur (AT)	`0040h:0016h`	1 octet

Format du mot en 0040h : 0010h

Numéro de bit	*Signification*
15-14	Nombre d'imprimantes
13-12	Réservés
11-9	Nombre d'adaptateurs asynchrones RS/232
8	Réservé
7-6	Nombre de lecteurs de disquettes (0 = 1)

(suite du tableau)

Numéro de bit	*Signification*
5-4	Mode vidéo au lancement : 0 : EGA/VGA 1 : 40x25 couleurs 2 : 80x25 couleurs 3 : 80x25 noir et blanc
3	Réservé
2	Périphérique de pointage (souris/crayon)
1	Coprocesseur mathématique (si = 1)
0	Disquette bootable

Tableau 2.2
Données concernant l'équipement.

Les adresses de ports sont des adresses de segment. Il y en a quatre par périphérique. Si le PC n'a qu'une imprimante, les trois valeurs qui suivent la première adresse sont sans signification.

Données concernant le clavier

Ces données renseignent essentiellement sur l'état actuel du clavier (touches "Shift", c'est-à-dire Ctrl, Alt, etc.). On y adjoint ici les données supplémentaires, normalement séparées :

Description	*Adresse*	*Type*
Touches "Shift"	0040h:0017h	1 octet (voir description)
Touches "Shift" du clavier étendu	0040h:0018h	1 octet (voir description)
Réservé. Alt et touches Num	0040h:0019h	1 octet
Adresse du caractère dans le buffer	0040h:001Ah	1 mot
Adresse de début du buffer	0040h:001Ch	1 mot
Buffer 16 octets	0040h:001Eh	16 octets (voir description)
Drapeau Ctrl-Break	0040h:0071h	1 octet
Début du buffer	0040h:0080h	1 mot
Offset de fin du buffer	0040h:0082h	1 mot
Octet d'état clavier	0040h:0096h	1 octet (voir description)
Octet d'état des LED	0040h:0097h	1 octet (voir description)

Format des octets concernant les touches "Shift" :

Numéro de bit	*Signification*	
(0040h:0017h)		
7	= 1	Insert Actif
6	= 1	Caps Lock actif
5	= 1	Num Lock actif
4	= 1	Scroll Lock actif
3	= 1	Alt pressé
2	= 1	Ctrl pressé
1	= 1	Shift Gauche pressé
0	= 1	Shift Droit pressé
(0040h:0018h)		
7	= 1	Insert pressé
6	= 1	Caps Lock pressé
5	= 1	Num Lock pressé
4	= 1	Scroll Lock pressé
3	= 1	Ctrl-Num Lock actif
2	= 1	Sys Req pressé
1	= 1	Alt gauche pressé
0	= 1	Ctrl droit pressé
(0040h:0071h)		
7	= 1	Ctrl-Break pressé
6..0		Réservés
(0040h:0096h)		
7	= 1	Lecture en cours ID
6	= 1	Dernier code = ID
5	= 1	Forcé à Num Lock
4	= 1	Clavier 101/102 touches
3	= 1	Alt droit actif
2	= 1	Ctrl droit actif
1	= 1	Dernier code = E0h
0	= 1	Dernier code = E1h

➥

(suite du tableau)

Numéro de bit		Signification
(0040h:0097h)		
7	= 1	Drapeau d'erreur de commande clavier
6	= 1	Mise à jour des LED en cours
5	= 1	Reçu un RESEND venant du clavier
4	= 1	Reçu un ACK venant du clavier
3	= 1	Réservé
2	= 1	LED Caps Lock allumée
1	= 1	LED Num Lock allumée
0	= 1	LED Scroll Lock allumée

Tableau 2.3
Données clavier.

Si l'adresse du prochain caractère à lire dans le buffer coïncide avec l'adresse de début du buffer, le buffer est vide.

Valeurs de Time-Out

Il y a une valeur de Time-Out par périphérique installable : une par imprimante et une par port RS/232. C'est généralement la même qui est répétée quatre fois.

Description	Adresse	Type
Time-Out des imprimantes	0040h:0078h	4 x 1 octet
Time-Out des ports RS/232	0040h:007Ch	4 x 1 octet

Tableau 2.4
Données Time-Out.

Données concernant les disquettes

Les données disquettes et les données supplémentaires sont particulièrement utiles au programmeur. Leur bon usage permet, entre autres, d'éviter de passer par les interruptions. Cependant, il faut faire attention : elles peuvent se révéler fausses, si l'on n'a pas encore accédé aux lecteurs.

Description	*Adresse*	*Type*
Octet d'état	0040h:003Eh	1 octet (voir description)
Etat des moteurs	0040h:003Fh	1 octet (voir description)
Valeur Time-Out	0040h:0040h	1 octet
Code de retour	0040h:0041h	1 octet
Octet d'état du contrôleur	0040h:0042h	6 octets
Vitesses de transmission des données	0040h:008Bh	1 octet (voir description)
Informations du contrôleur	0040h:008Fh	1 octet (voir description)
Type de média	0040h:0090h	2 x 1 octet
Service	0040h:0092h	2 x 1 octet
Numéro de piste	0040h:0094h	2 x 1 octet

Format des octets

Numéro de bit	*Signification*	
(0040h:003Eh)		
7	= 1	Une interruption matérielle disquette a eu lieu
6-4		Inutilisés
3-2		Réservés
1	= 1	Lecteur 1 recalibré
0	= 1	Lecteur 0 recalibré
(0040h:003Fh)		
7	= 1	Opération courante = Ecriture/Formatage
	= 0	Opération courante = Lecture/Vérification
6		Réservé
5-4	= 00	Drive 0 sélectionné
	= 01	Drive 1 sélectionné (les valeurs 10 et 11 sont réservées)

(suite du tableau)

Numéro de bit		Signification
3-2		Réservés
1	= 1	Moteur du lecteur numéro 1 en route
0	= 1	Moteur du lecteur numéro 0 en route
(0040h:008Bh)		
7-6		Dernière vitesse utilisée par le contrôleur : 00 = 500 Kb/s (KiloBits / Seconde) 01 = 300 Kb/s 10 = 250 Kb/s
5-4		Dernier ratio
3-2		Vitesse au début de l'opération (voir bits 7-6)
1-0		Réservés
(0040h:008Fh)		
7		Réservé
6	= 1	Lecteur 1 reconnu
5	= 1	Lecteur 1 supporte plusieurs vitesses
4	= 1	Lecteur 1 supporte plusieurs formats
3		Réservé
2	= 1	Lecteur 0 reconnu
1	= 1	Lecteur 0 supporte plusieurs vitesses
0	= 1	Lecteur 0 supporte plusieurs formats
(0040h:0090h et 0040h:0091h)		
7-6		Vitesse de transfert (voir 0040h:008Bh)
5		Disquette 360 Ko dans lecteur 1,2 Mo
4		Disquette présente dans le lecteur reconnue
3		Réservé
2-0		Type du média : 111 720 Ko dans lecteur 720 Ko ou 1,44 Mo dans lecteur 1,44 Mo 101 1,2 Mo dans lect 1,2 Mo 100 360 Ko dans lect 1,2 Mo 011 360 Ko dans lect 360 Ko 010 Essai 1,2 Mo dans lecteur 1,2 Mo 001 Essai 360 Ko dans lecteur 1,2 Mo 000 Essai 360 Ko dans lecteur 360 Ko

Tableau 2.5
Données disquettes.

Les codes d'erreur éventuels renvoyés lors de la dernière opération du lecteur de disquette sont stockés en 0040h:0041h. Leur signification est donnée au chapitre 4 (*Disques au niveau physique : gestion par le BIOS*). Les informations contenues aux adresses d'offset 0090h, 0092h et 0094h sont chacune sur un octet. Le suivant stocke les valeurs correspondantes pour le lecteur 1.

Données sur le(s) disque(s) fixe(s)

Ces données, au même titre que celles qui s'intéressent aux disquettes, sont essentielles au bon fonctionnement du PC. Mais attention, le disque *fixe* n'est pas le disque *dur* : une subtile distinction s'établit en effet entre le matériel proprement dit (un disque externe ou interne) et le logiciel de bas niveau (MS-DOS), qui ne peut gérer qu'une partie de ce matériel (32 Mo jusqu'à la version 3.3, 2 048 Mo depuis la version 4.0).

Le système d'exploitation partitionne donc un disque fixe en plusieurs disques durs : le disque fixe de l'auteur ayant une contenance de 71 Mo, il a été partitionné en trois unités logiques (C:, D:, E:) qui sont autant de disques durs. Le BIOS, lui, ne fait pas ce genre de distinctions et ne s'intéresse qu'au matériel : c'est pourquoi on parle ici de disque fixe (comme le font d'ailleurs les divers ouvrages de référence techniques du BIOS).

Description	***Adresse***	***Type***
Code de retour	0040h:0074h	octet
Nombre de disques fixes	0040h:0075h	octet
Octet de contrôle	0040h:0076h	octet
Offset du port de disque fixe	0040h:0077h	octet
Registre d'état (AT)	0040h:008Ch	octet
Registre d'erreur (AT)	0040h:008Dh	octet
Drapeau d'interruption (AT)	0040h:008Eh	octet

Tableau 2.6
Données disques fixes.

Le code de retour de la dernière opération effectuée par le disque dur est stocké en 0040h:0074h. Il correspond aux codes d'erreur dont la table se trouve au chapitre 4 (*Disques au niveau physique : gestion par le BIOS*).

Timer

Au sujet des données de l'horloge entreposées par le BIOS, on ne peut dire que deux choses : elles sont peu nombreuses (trois seulement), et utiliser l'Int `1Ah` est nettement plus simple que de passer par elles.

Description	***Adresse***	***Type***
Nombre de tics après minuit	`0040h:006Ch`	4 octets
24 heures dépassées ?	`0040h:0070h`	1 octet
Nombre de jours depuis le 1:1:80	`0040h:00CEh`	1 octet

Tableau 2.7
Données Timer.

Données vidéo

Les données vidéo du BIOS sont probablement les plus utilisées par les programmes d'application.

Description	***Adresse***	***Type***
Mode vidéo courant	`0040h:0049h`	octet
Nombre de colonnes	`0040h:004Ah`	mot
Taille de la page active	`0040h:004Ch`	mot
Adresse de la page active	`0040h:004Eh`	mot
Position du curseur	`0040h:0050h`	4 x 2 octets
Type du curseur	`0040h:0060h`	2 octets
Numéro de la page active	`0040h:0062h`	octet
Numéro du port vidéo	`0040h:0063h`	mot
Registre de sélection du mode courant	`0040h:0065h`	octet
Valeur palette active	`0040h:0066h`	octet
Nombre de lignes (VGA)	`0040h:0084h`	octet
Taille des caractères (VGA)	`0040h:0085h`	mot

(suite du tableau)

Description	Adresse	Type
Octet de contrôle (VGA)	0040h:0087h	octet (voir description)
Switch données (VGA)	0040h:0088h	octet (voir description)
Octet de contrôle (VGA)	0040h:0089h	octet (voir description)
Index dans table DCC (VGA)	0040h:008Ah	octet

Format des octets

Numéro de bit		Signification
(0040h:0087h)		
7	= 1	RAM vidéo présente
6-5		Taille mémoire de l'adaptateur : 00 = 64 Ko 01 = 128 Ko 10 = 92 Ko 11 = 256 Ko
4		Inutilisé
3	= 0	Adaptateur EGA/VGA actif
2	= /	Attente à l'affichage
1	= 1	Moniteur monochrome, adaptateur EGA/VGA
	= 0	Moniteur couleurs ou ECD
0	= 1	Pas de translation curseur avec un moniteur ECD en mode 350 lignes
	= 0	Translation curseur
(0040h:0088h)		
7-4		Bits 3-0 du connecteur
3-0		Switches 3-0
(0040h:0089h)		
7	= 1	200 lignes
6-5		Réservés
4	= 1	400 lignes
3	= 1	Pas de palette chargée
2	= 1	Moniteur monochrome
1	= 1	Echelle de gris
0		Réservé

Tableau 2.8
Données vidéo.

La position du curseur est stockée pour chaque page écran (quatre). Le premier octet est le numéro de colonne du curseur dans la page, le second le numéro de ligne : la position 0,0 correspond à un GotoXy(1,1) en Turbo Pascal. Le type du curseur définit ses numéros de ligne scan de début et de fin. Le premier octet est le numéro de la ligne de fin, le second le numéro de ligne de début. Un curseur "classique", comme celui du DOS en mode texte 80x25, débute en ligne 6 et se termine en ligne 7. Attention cependant : les cartes *Hercules* ne respectent pas cette norme. Si vous modifiez ces valeurs, sauvez d'abord les valeurs originales. Le numéro du port vidéo est toujours 03D4h ou 03B4h (B pour monochrome, D pour couleurs). La plupart du temps, les valeurs marquées VGA seulement sont également valables avec une carte EGA. Enfin, si vous modifiez une des données BIOS vidéo, la modification ne prendra effet qu'à la prochaine Int 10h.

Données diverses

Ces données n'entrent dans aucune des précédentes rubriques, ou bien il faudrait faire une rubrique par donnée. On les réunira donc sous cette dénomination générique.

Description	***Adresse***	***Type***
Offset du drapeau d'attente (AT)	0040h:0098h	mot
Segment du drapeau d'attente (AT)	0040h:009Ah	mot
Compteur d'attente LSW (AT)	0040h:009Ch	mot
Compteur d'attente MSW (AT)	0040h:009Eh	mot
Drapeau d'attente actif	0040h:00A0h	octet (voir description)
Réservé aux réseaux	0040h:00A1h	7 octets
Pointeur sur les paramètres vidéo (VGA)	0040h:00A8h	2 mots
Réservés	0040h:00B0h	6 octets
Réservés	0040h:00C0h	14 octets
Réservés	0040h:00CFh	48 octets
Octet d'état de l'impression écran	0040h:0100h	1 octet

Format de l'octet drapeau d'attente actif :

Numéro de bit	Signification	
(0040h:00A0h)		
7	= 1	Temps d'attente écoulé
6-1		Réservés
0	= 1	Il y a eu une Int 15h, fonction 86h

Tableau 2.9
Données diverses.

Tout ce qui concerne le compteur d'attente utilisateur est réservé aux AT. Le compteur lui-même est stocké en deux mots, dont le moins significatif (*Least Significant Word*) se trouve à l'adresse la plus basse (009Ch contre 009Eh), ce qui correspond aux normes Intel de gestion de la mémoire. L'octet d'impression de l'écran indique si l'Int 05h est en train de se dérouler.

BiosData.Pas : interpréter les données BIOS en RAM

Le programme BiosData.Pas est fourni sur la disquette d'accompagnement. Sa longueur (environ 800 lignes) fait qu'il n'est pas reproduit ici. BiosData.Pas a pour but d'afficher en clair toutes les données du BIOS présentes en RAM, avec leur signification. Il appelle donc surtout deux types de procédures : conversion en hexadécimal ou en binaire (lorsque les données sont à afficher bit à bit), et affichage. Cela le rend assez monotone. Il n'empêche que son utilité est réelle : il est beaucoup plus agréable d'emploi que n'importe quel *dumper* et fournit la signification de chaque donnée affichée. Les données sont regroupées par thème selon le classement opéré par IBM, excepté en ce qui concerne les "données diverses" qui réunissent plusieurs rubriques.

Chaque thème fait l'objet d'une procédure, appelée par l'intermédiaire de la boucle principale de choix d'une option à l'intérieur du menu. Certaines procédures sont imbriquées : elles ont généralement pour tâche d'afficher une chaîne en réponse à une valeur (comme dans le cas de la taille mémoire de l'adaptateur vidéo, où il faut afficher '64 Ko' si les bits 6 et 5 de l'octet en 0040h:0087h sont à zéro). Attention : l'auteur a pu oublier certains tests à effectuer sur le modèle du PC (IF AT THEN...). Il est conseillé de vérifier et de les rajouter, si c'est nécessaire. Sans quoi, les données obtenues sur PC dans une rubrique réservée aux AT n'auraient aucun

sens. Des tests sont également à prévoir sur la carte vidéo (qui doit être au moins une EGA). Dernière observation avant de passer à la table des références croisées, `BiosData.Pas` fait souvent appel à l'unité `SYS.TPU`, dont le code est listé en annexe : reportez-vous y si besoin était.

Procédure	***Description***	***Ligne***
`TraitsOctet`	Affiche 1 flèche/bit	*5*
`Equipement`	Affiche les données équipement	*15*
`Video`	Mode vidéo au lancement	*18*
`DonneesClavier`	Affiche les données clavier	*87*
`DonneesDisquette`	Affiche les données disquettes	*149*
`LectEcr`	Voir `0040h:003Fh`, bit 7	*152*
`Erreur`	Nom d'erreur selon son numéro	*160*
`Video`	Affiche les données vidéo	*233*
`DonneesPOST`	Affiche les données POST	*268*
`DonneesTimer`	Affiche les données timer	*307*
`DonneesDisqueDur`	Affiche les données du disque fixe	*323*
`Erreur`	Nom d'erreur selon son numéro	*325*
`ValeursTimeOut`	Affiche les données Time-Out	*372*
`InfosEtenduesClavier`	Affiche les données étendues clavier	*393*
`InfosEtenduesVideo`	Affiche les données étendues vidéo	*445*
`InfosEtenduesDisqueDur`	Affiche les données étendues du disque fixe	*508*
`InfosEtenduesDisquette`	Affiche les données étendues de la disquette	*524*
`Format`	`0040h:0090h`, bits `2-0`	*526*
`Vitesse`	`0040h:008Bh`, bits `7-6`	*539*
`DiversesDonneesBios`	Affiche les données diverses	*611*
`ChoisitOption`	Lit une touche dans le menu	*653*
`Curseur`	Affiche/éteint curseur	*672*
`Selectionne`	Choisit la procédure à appeler	*677*

Tableau 2.10
Références croisées de BiosData.Pas. (listing disponible sur la disquette d'accompagnement).

Conclusion

Nous avons vu en détail quelles sont les données que le BIOS entrepose en RAM. Le programme `BiosData.Pas` permet d'examiner leur contenu.

Chapitre 3

RAM gérée par le DOS

Mots-clefs

Buffers Le DOS tient à jour en permanence une liste de tampons (ou *buffers*) qui contiennent les derniers secteurs lus sur le disque. La plupart du temps, ce sont la FAT et le contenu du répertoire courant qui sont ainsi mémorisés.

DBP Ce sont les blocs de paramètres disque (ou *disk block parameters*). Il s'agit d'une structure de données qui mémorise des informations vitales concernant chaque disque, comme le numéro du premier secteur de données, ou celui de la FAT, le nombre de secteurs par cluster, etc.

Device drivers Appelés aussi pilotes de périphériques, ces programmes servent de liaison entre le système d'exploitation et les périphériques matériels (l'écran, les disques, les ports séries et parallèles, etc.). Certains sont chargés par le fichier IBMBIO.COM (IO.SYS) au lancement du PC. D'autres peuvent être installés par l'utilisateur à condition qu'ils soient déclarés dans le fichier CONFIG.SYS.

Fonction EXEC Le chargeur du DOS n'est autre que la fonction 4Bh de l'Int 21h. Elle a pour mission de charger et d'exécuter un programme ou de charger un overlay.

MCB Les blocs de contrôle de la mémoire (ou *memory control blocks*) sont une structure mémorisant le contenu d'un bloc de mémoire, ses adresses de début et de fin et son type (code ou données). Ils permettent de savoir si la mémoire est fragmentée, s'il en reste de disponible, etc.

Mémoire système Nous appelons *mémoire système* la RAM employée par le DOS et ses données. Le reste est appelé mémoire *utilisateur*.

Nœud d'informations En informatique, un *nœud d'informations* est une structure de données conçue comme une table de hachage, qui regroupe les adresses de plusieurs données essentielles mais de genre différent. La fonction 52h de l'Int 21h (non documentée par Microsoft) renvoie ainsi un nœud d'informations sur la plupart des structures DOS étudiées dans ce chapitre.

PSP Le préfixe de segment de programme (ou *program segment prefix*) est une structure de données mémorisant les renseignements dont un programme utilisateur pourrait avoir besoin, et qui ne se trouvent pas ailleurs. Nous verrons plusieurs de ses champs réservés.

Table des chemins Le DOS maintient en mémoire une table qui contient le chemin courant de chaque disque.

Table des vecteurs d'interruption L'adresse en mémoire de chaque interruption se trouve mémorisée dans une table en 0000h:0000h. C'est ce qui permet de détourner un vecteur d'interruption.

La mémoire vive (RAM) est au cœur du micro-ordinateur : sans elle, il serait absolument impossible d'exécuter un programme, quel qu'il soit. Plus l'informatique se perfectionne et plus la RAM devient vitale : Windows 3.0, OS/2, les logiciels les plus courants (qu'il s'agisse de *Paradox* ou du dernier compilateur C de Microsoft ou de Borland) sont autant d'applications qui exigent d'importantes quantités de mémoire pour fonctionner correctement. On a beau utiliser les techniques les plus pointues pour essayer de les faire tenir dans les misérables 640 Ko supportés par le DOS, la LIM EMS et la mémoire étendue sont déjà devenues des passages obligés.

Bien que ce chapitre ne traite pas des méthodes de gestion de l'EMS ou de l'XMS, il revêt une importance certaine dans l'ouvrage puisqu'il explique comment le DOS gère la mémoire de base. Ce n'est qu'après avoir bien compris ces principes qu'il vous sera possible de travailler sur la mémoire étendue et paginée.

Le système d'exploitation doit absolument pouvoir distinguer un programme des données lorsque les deux sont en mémoire : sans quoi, aucun programme ne pourrait jamais s'exécuter. Or le micro-ordinateur ne connaît qu'une seule manière (le binaire) de coder l'information *quelle qu'en soit la nature.* Comment donc décider que la séquence d'octets qui se trouve en `0000h:0000h` représente une donnée, tandis que celle qui se trouve en `07C0h:0100h` représente un programme à interpréter ? La manière la plus simple d'opérer ces distinctions est de mémoriser le type (code ou données) d'un élément et son adresse *avant* de le charger en mémoire. Il faut pour cela disposer des structures de données chargées de cette mémorisation et ne pas se tromper sur leur propre emplacement mémoire. Nous étudierons ces structures de données – les MCB – dans ce chapitre.

Mais d'autres difficultés surgissent lorsque l'on souhaite organiser la mémoire vive. Tout d'abord, le système d'exploitation doit lui-même se trouver en mémoire pour fonctionner. Il possède ses données propres, qui doivent également se trouver en mémoire. Enfin, comme sa tâche est de gérer l'ensemble du micro-ordinateur, il doit accéder rapidement aux device drivers, ainsi qu'aux buffers de disques et à diverses informations relatives aux périphériques : ces données doivent par conséquent se trouver en mémoire. Il y a donc une partie de la RAM réservée au DOS en plus de celle réservée par le BIOS. L'espace utilisateur s'en trouve réduit d'autant. En outre, contrairement au cas du BIOS, les interruptions DOS se trouvent en RAM et non en ROM. Tous ces éléments gagnent à être connus si l'on veut étudier la mémoire et les systèmes d'exploitation de près.

Ces contraintes permettent de comprendre les problèmes incessants que rencontre tout programmeur souhaitant optimiser l'utilisation que fait son programme de la mémoire, en même temps qu'elles expriment l'absolue nécessité de procéder à de telles optimisations.

Mémoire système

La configuration de la mémoire après le lancement du micro-ordinateur et le chargement du DOS est définitive en ce qui concerne la RAM occupée par le système d'exploitation et ses structures de données. En revanche, la partie utilisateur est constamment modifiée, y compris les structures de données mises en place par le DOS pour en contrôler l'attribution mémoire.

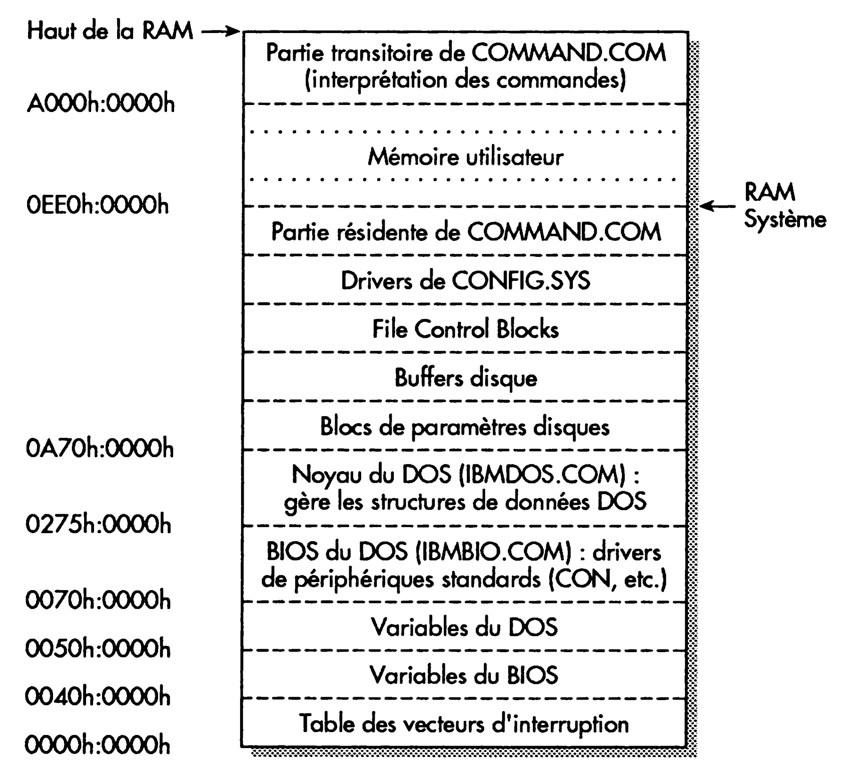
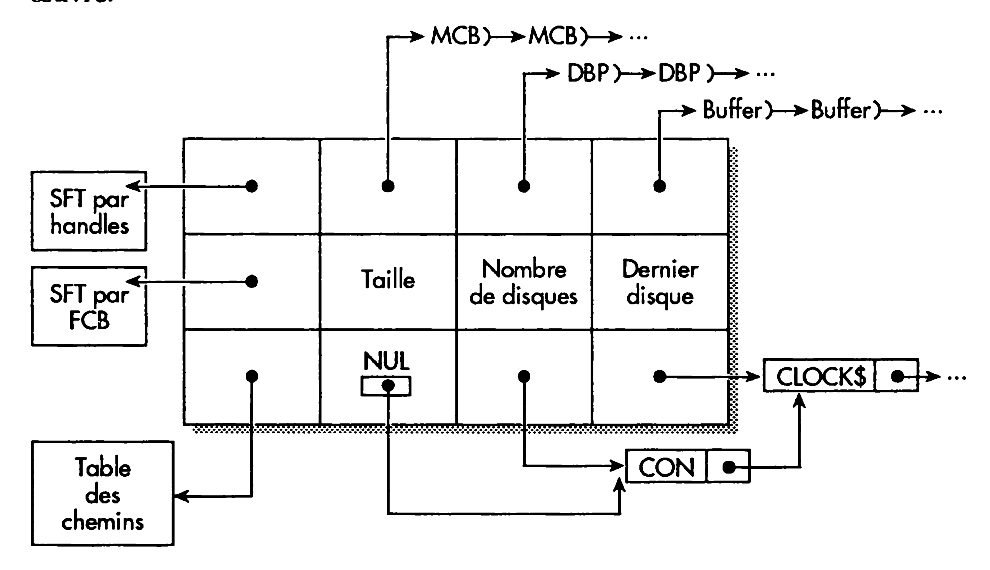


Figure 3.1
Carte schématique de la RAM (système et utilisateur).

Remarque — Les adresses au-dessus de `0070h` sont approximatives ainsi que l'ordre dans lequel apparaissent les blocs de paramètres disque, les buffers, les FCB et les drivers installés par l'utilisateur.

Organisation des structures de données entre elles

Le principal problème posé par cette organisation de la mémoire tient au fait qu'un exemplaire de chaque structure de données est nécessaire par objet à représenter et que le nombre d'objets représentés est théoriquement infini. Ainsi, le DOS déclare à lui seul 14 pilotes de périphériques (NUL, COM, AUX, PRN, CLOCK$, pilote disque, COM1, LPT1, LPT2, LPT3, COM2, COM3, COM4). Mais le fichier CONFIG.SYS permet à l'utilisateur d'en charger de nouveaux qui s'ajouteront à cette liste, dont il est impossible de connaître la taille à l'avance. C'est pourquoi chaque structure de donnée employée utilise le principe des listes chaînées de pointeurs, type de données qui permet une extension mémoire quasiment infinie. Encore est-il nécessaire de savoir où trouver le premier objet de la liste pour pouvoir consulter les autres. Le DOS utilise une fonction de l'Int 21h (Int 21h, fonction 52h, non documentée) pour remonter le cours des différentes listes. En sortie, cette interruption fournit en effet l'adresse de la cellule de tête de chaque liste. C'est le principe des *nœuds d'information* (notamment utilisé par UNIX pour gérer les fichiers) qui est mis en œuvre.



On distingue trois têtes de liste chaînées, trois pointeurs sur des tables, trois champs de données, deux pointeurs sur des device drivers et le premier device driver de la liste (NUL) à laquelle les device drivers pointés appartiennent.

Figure 3.2
Fonctionnement schématisé du nœud d'informations du DOS.

Le nœud d'informations contient six champs. Chacun d'eux pointe sur une cellule, qui appartient souvent à une liste chaînée.

ES:BX-04h	Offset du premier MCB
ES:BX-02h	Segment du premier MCB
ES:BX	Offset du premier DBP
ES:BX+02h	Segment du premier DBP
ES:BX+04h	Offset de la SFT handles
ES:BX+06h	Segment de la SFT handles
ES:BX+08h	Offset du device driver CLOCK$
ES:BX+0Ah	Segment du device driver CLOCK$
ES:BX+0Ch	Offset du device driver CON
ES:BX+0Eh	Segment du device driver CON
ES:BX+10h	Taille maximale d'un secteur
ES:BX+12h	Offset du premier buffer disque
ES:BX+14h	Segment du premier buffer disque
ES:BX+16h	Offset de la table des chemins
ES:BX+18h	Segment de la table des chemins
ES:BX+1Ah	Offset de la SFT FCB
ES:BX+1Ch	Segment de la SFT FCB
ES:BX+1Eh	Nombre de drives
ES:BX+20h	Numéro du dernier drive
Le Driver 'NUL' commence ici	
ES:BX+22h	Offset du prochain en-tête de driver
ES:BX+24h	Segment du prochain en-tête de driver
ES:BX+26h	Attribut du driver NUL (8004h)
ES:BX+28h	Offset de la routine de stratégie de NUL
ES:BX+2Ah	Offset de la routine de l'interruption de NUL
ES:BX+2Ch	Nom du driver ('NUL— — — — — ')

Figure 3.3
Format du nœud d'information du DOS (fonction 52h, Int 21h).

Si les principes qui régissent cette organisation sont presque parfaits, l'application qu'en fait le DOS est beaucoup plus discutable. On en fournira une illustration avec les drivers : ceux-ci, nous l'avons dit, sont regroupés dans une liste dont la

première cellule est le pilote NUL. Une manière à la fois sûre et élégante de permettre l'accès aux autres pilotes aurait été que le nœud d'information contienne un pointeur sur NUL. Au lieu de cela, NUL fait partie du nœud d'information lui-même : il se trouve au segment ES et à l'offset BX+22h (voir page ci-contre).

La fonction 52h de l'Int 21h renvoie à partir de ES:(BX-4) jusqu'à ES:(BX+1Eh) une série de pointeurs sur les premières occurrences des structures de données DOS de gestion de la mémoire et des périphériques (voir le schéma sur les principes des nœuds d'information). Nous étudions certaines de ces structures de données dans ce chapitre.

Listing 3.4

Lecture du nœud d'informations du DOS. ❶

```
PROGRAM VerifieNoeudInfo;  { InfoDos.Pas }

USES Dos, Sys;

TYPE
  Dev     =  RECORD
               NextHeaderOfs,
               NextHeaderSeg,
               Attribut,
               StrategyOfs,
               InterruptOfs    : Word;
               Nom             : ARRAY[0..7] Of Char;
             END;
   Noeud  =  RECORD
                McbOfs, McbSeg,
                DcbOfs, DcbSeg,
                HdlSFTOfs, HdlSFTSeg,
                ClockOfs, ClockSeg,
                ConOfs, ConSeg,
                MaxSect,
                BufOfs, BufSeg,
                CDSOfs, CDSSeg,
                FcbSftOfs, FcbSftSeg,
                FcbSftTail,
                NbDrv, LastDrv          : Word;
                NUL                     : Dev;
             END;

VAR  Info : Noeud;
     Ok   : BOOLEAN;

FUNCTION LitInfoDos : BOOLEAN;
VAR Regs : Registers;
```

➥

Lecture du nœud d'informations du DOS (suite). ❷

```
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $52;
    MsDos(Regs);
    IF (FLAGS AND 1 = 1) THEN
      LitInfoDos := FALSE
    ELSE
    BEGIN
      WITH Info DO
      BEGIN
        McbOfs := MemW[Es:(Bx-4)];
        McbSeg := MemW[Es:(Bx-2)];
        DcbOfs := MemW[Es:Bx];
        DcbSeg := MemW[Es:Bx+2];
        HdlSFTOfs := MemW[Es:Bx+4];
        HdlSFTSeg := MemW[Es:Bx+6];
        ClockOfs := MemW[Es:Bx+8];
        ClockSeg := MemW[Es:Bx+$A];
        ConOfs := MemW[Es:Bx+$C];
        ConSeg := MemW[Es:Bx+$E];
        MaxSect  := MemW[Es:Bx+$10];
        BufOfs := MemW[Es:Bx+$12];
        BufSeg := MemW[Es:Bx+$14];
        CDSOfs := MemW[Es:Bx+$16];
        CDSSeg := MemW[Es:Bx+$18];
        FcbSftOfs := MemW[Es:Bx+$1A];
        FcbSftSeg := MemW[Es:Bx+$1C];
        FcbSftTail := MemW[Es:Bx+$1E];
        NbDrv := Mem[Es:Bx+$20];
        LastDrv := Mem[Es:Bx+$21];
        WITH NUL DO
        BEGIN
          NextHeaderOfs := MemW[Es:Bx+$22];
          NextHeaderSeg := MemW[Es:Bx+$24];
          Attribut      := MemW[Es:Bx+$26];
          StrategyOfs   := MemW[Es:Bx+$28];
          InterruptOfs  := MemW[Es:Bx+$2A];
          Move(Mem[Es:Bx+$2C], Nom, 8);
        END;
      END;
      LitInfoDos := TRUE;
    END;
  END;
End;
```

➥

Lecture du nœud d'informations du DOS (suite). ❸

```
Begin
  Ok := LitInfoDos;
  IF Ok THEN
  BEGIN
    WriteLn;
    WITH Info DO
    BEGIN
      WriteLn('Premier MCB en : ',
               MotDecVersHex(McbSeg),':',
               MotDecVersHex(McbOfs));
      WriteLn('DCB en : ', MotDecVersHex(DcbSeg), ':',
               MotDecVersHex(DcbOfs));
      WriteLn('SFT Handles en : ',
               MotDecVersHex(HdlSFTSeg), ':',
               MotDecVersHex(HdlSFTOfs) );
      WriteLn('CLOCK$ en : ', MotDecVersHex(ClockSeg),
              ':', MotDecVersHex(ClockOfs) );
      WriteLn('CON en : ', MotDecVersHex(ConSeg), ':',
              MotDecVersHex(ConOfs));
      WriteLn('longueur maximale d''un secteur : ',
              MaxSect);
      WriteLn('Buffers en : ',
              MotDecVersHex(BufSeg),':',
              MotDecVersHex(BufOfs));
      WriteLn('Table des chemins en : ',
              MotDecVersHex(CDSSeg), ':',
              MotDecVersHex(CDSOfs) );
      WriteLn('SFT FCB en : ',
              MotDecVersHex(FcbSftSeg),':',
              MotDecVersHex(FcbSftOfs));
      WriteLn('Taille de la SFT FCB = ',
              MotDecVersHex(FcbSftTail));
      WriteLn('Nbre de drives : ', NbDrv);
      WriteLn('Dernier drive : ', LastDrv);
      WITH NUL DO
      BEGIN
        WriteLn('Device =  ', Nom);
        WriteLn(#9,'Prochain Header de Device en ',
                MotDecVersHex(NextHeaderOfs),':',
                MotDecVersHex(NextHeaderSeg));
        WriteLn(#9,'Attribut    : ',
                MotDecVersHex(Attribut));
        WriteLn(#9,'Stratégie à l''Offset : ',
                MotDecVersHex(StrategyOfs));
```

➥

Lecture du nœud d'informations du DOS (suite). ❹

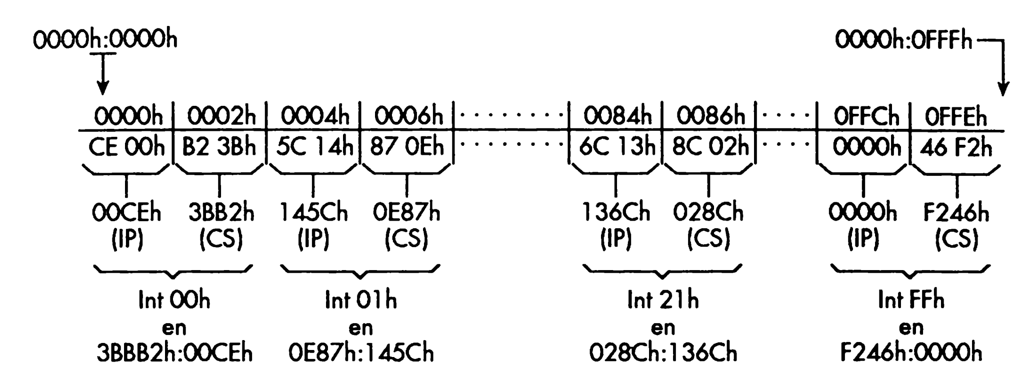
```
        WriteLn(#9, 'Interruption à l''Offset : ',
                MotDecVersHex(InterruptOfs));
      END;
    END;
  END
  ELSE
  BEGIN
    WriteLn(' Abandon');
    Halt(1);
  END;
End.
```

Examen des structures de données

L'une des premières choses que fait le DOS au lancement du PC est d'initialiser une partie de la table des vecteurs d'interruptions. Les numéros d'interruptions DOS sont compris entre `20h` et `3Fh`. Parmi celles-ci, les interruptions `28h` à `3Fh` ainsi que les fonctions `18h`, `1Dh` à `20h`, `32h`, `34h`, `37h`, `50h` à `53h`, `55h`, `5Dh`, `60h`, `61h`, `63h`, `64h`, `69h` à `7Fh` de l'Int `21h` sont réservées (voir l'*Annexe* 2 pour plus de détails sur les fonctions réservées). L'intérêt principal de la table des vecteurs d'interruptions est de fournir l'adresse mémoire à laquelle se trouve l'Int recherchée : il suffit ensuite de savoir utiliser DEBUG pour comprendre le fonctionnement d'une interruption, voire, avec de la patience, d'une fonction de l'Int `21h`. Les routines d'interruptions étant logées en RAM, on pourrait aller jusqu'à les modifier (ce qui est fortement déconseillé).

Table des vecteurs d'interruption

Le format de la table des vecteurs d'interruptions est bien connu : chaque enregistrement de cette table contient deux mots. Le premier mot est l'adresse de déplacement de l'Int, le second mot est son adresse de segment. Un enregistrement étant long de quatre octets (deux fois deux mots), pour accéder à l'enregistrement d'une Int particulière on multiplie son numéro d'ordre par quatre. Le résultat de la multiplication fournit l'index de déplacement dans la table auquel on trouvera l'adresse de l'Int.



La table commence en 0000h:0000h et se termine en 0000h:0FFFh. Les valeurs de segment et déplacement sont inversées. Elles sont stockées à l'index (NoInt x 4) où NoInt représente le numéro de l'interruption. Aussi, peut-on lire les valeurs de segment et déplacement de l'Int 21h à partir de l'octet 84h (4 x 21h = 84h) jusqu'à l'octet 87h.

Sous DEBUG, on fera :

```
-d 0000:0084 1 4        ; 4 octets à (21h * 4) = 84h
0000:0080                  77  14  75  02
-u 0275:1477            ; Désassemblage du début de l'Int 21h
0275:1477 2E              CS:
0275:1478 3A26FF0D        CMP  AH,[0DFF]
0275:147C 77DC            JA   145A
0275:147E 80FC51          CMP  AH,51
0275:1781 74A1            JZ   1424
```

Figure 3.5
Organisation de la table des vecteurs d'interruptions.

La table contient les adresses de toutes les routines d'interruptions (matérielles, BIOS et DOS). Le DOS initialise celles qui le concernent au moment de son chargement. La table se trouve en 0000h:0000h, fait 1 Ko (256 vecteurs * 4 octets) et se termine en 0000h:3FFFh. On peut en examiner le contenu avec DEBUG et désassembler les routines d'interruption.

Listing 3.6
Programme MapInt.Pas. ❶

```
PROGRAM ListeAdressesInt;  { MapInt.Pas }

USES Crt, Sys;

VAR  Segment, Offset : Word;
     TabInt          : ARRAY[0..511] OF Word;
```

➥

Programme MapInt.Pas (suite). ❷

```
PROCEDURE TableInterruptions;
VAR i,j : Integer;
BEGIN
  i:=0; j:=0;
  WHILE (j <= 255) DO
  BEGIN
    TabInt[i]   := MemW[0:(j*4)+2)]; { CS }
    TabInt[i+1] := MemW[0:(j*4)];    { IP }
    Inc(j); Inc(i,2);
  END;
END;

PROCEDURE ListeInt;
VAR i, Lig, Col : Integer;
BEGIN
 i:=0; ClrScr; Lig:=4; Col:=6; GotoXy(1,2);
 Write('INT   0      1      2      3      4       5');
 GotoXy(3,4); Write('0 | ');
 WHILE (i <= 511) DO
 BEGIN
   IF (Lig = 24) THEN
   BEGIN
     Write('----- Appuyez sur <┘ ------');
     ReadLn; ClrScr; Lig := 4; Col:=6; GotoXy(1,2);
     Write('INT    0      1      2      3     4      5');
     GotoXy(3,4); Write('0 | ');
   END;
   IF (Col < (80-11)) THEN
     GotoXy(Col, Lig)
   ELSE
   BEGIN
     GotoXy(1, Lig+1); Write((i DIV 2):3, ' | ');
     GotoXy(6, Lig+1);
   END;
   Write(' ', MotDecVersHex(TabInt[i]), ':');
   Write(MotDecVersHex(TabInt[i+1]), ' ');
   Inc(i,2); Lig:=WhereY; Col:=WhereX;
 END;
END;

BEGIN
  TableInterruptions;
  ListeInt;
END.
```

Accéder aux pilotes de périphériques

Après avoir installé ses routines d'interruptions, le DOS initialise chacun des pilotes de périphériques standard. Ceux-ci ont été chargés en mémoire lors de la lecture du fichier IBMBIO.COM (IO.SYS). Pour effectuer cette initialisation, le DOS remonte la file des drivers en commençant par le premier (NUL). Lorsqu'il a exécuté la routine d'initialisation d'un driver, il passe au suivant, dont l'adresse lui est indiquée par un champ du driver actuel. C'est donc bien à une liste chaînée de pointeurs que l'on a affaire.

Un device driver se décompose en trois parties principales : un en-tête, une routine de stratégie et une routine d'interruption. C'est l'en-tête qui fait le lien avec les routines qui le suivent (stratégie et interruption), ainsi qu'avec le prochain driver.

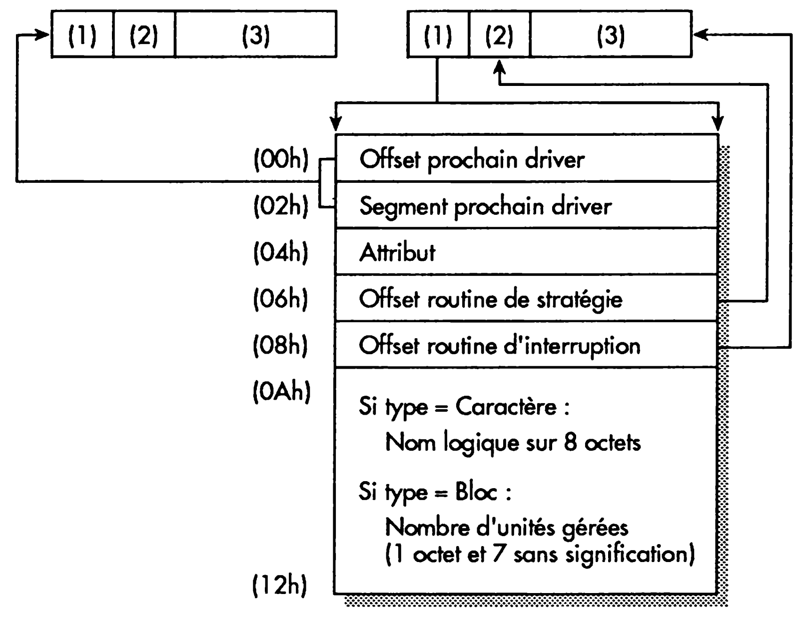


Figure 3.7
En-tête d'un pilote de périphérique.

Un pilote de périphérique se compose de trois parties :

1. l'en-tête, ici schématisé ;
2. la routine de stratégie dont l'adresse se trouve au champ 06h de l'en-tête ;
3. la routine d'interruption, dont l'adresse est au champ 08h.

Les deux premiers champs d'un en-tête de driver permettent de trouver le pilote suivant de la liste. Le mot d'attribut (champ 04h de l'en-tête de driver) donne plusieurs informations sur le comportement du pilote. Le bit 15 indique s'il s'agit d'un dispositif bloc (bit à 0) ou caractère (bit à 1). Le bit 2 identifie le driver NUL s'il est à 1. Les treize autres bits concernent la mise en œuvre du driver.

Le principal intérêt d'afficher la liste des drivers tient à ce que l'on peut vérifier si un pilote de périphérique en a remplacé un autre. En effet, lorsque l'utilisateur déclare un driver dans le fichier CONFIG.SYS, celui-ci remplace le plus souvent un des pilotes standard. Par exemple, ANSI.SYS (pilote d'écran amélioré fourni avec le DOS), en s'ajoutant à la liste des pilotes standard, court-circuite l'accès à CON : il porte le même nom logique et s'installe une cellule avant dans la liste. On comprend donc que le DOS, lorsqu'il fait appel à CON passe en fait par ANSI.SYS, puisqu'il effectue sa recherche à partir de la source (seule cellule dont il connaisse directement l'adresse).

Pour remonter la liste des drivers, on exécute une Int 21h, fonction 52h : le driver NUL se trouve en ES:(BX+22h). On vérifie qu'il s'agit bien de NUL grâce au mot d'attribut et/ou au nom logique (NUL est un dispositif Caractère). Si c'est bien le cas, on cherche le pilote suivant à l'aide de ses deux premiers champs. Sinon, on fait de nouveau appel à l'Int 21h, fonction 52h pour trouver l'adresse du driver CON et l'on poursuit la recherche. On aura cependant perdu l'adresse de NUL et des éventuels drivers qui se trouvent entre NUL et CON.

Deux configurations de drivers en RAM

1. Configuration standard

Driver	*Type*	*Nom*
0275:0048	C	NUL
0070:016E	C	CON
0070:0180	C	AUX
0070:0192	C	PRN
0070:01A4	C	CLOCK$
0070:01B6	B	5 unités logiques (A: à E:)
0070:01CA	C	COM1
0070:01DC	C	LPT1
0070:01EE	C	LPT2
0070:0200	C	LPT3
0070:0212	C	COM2
0070:0224	C	COM3
0070:0236	C	COM4

2. Avec, dans CONFIG.SYS :

   DEVICE=C:\Dos\QUEMM.SYS (gère la LIM EMS)

   DEVICE=C:\Dos\ANSI.SYS

   DEVICE=C:\Dos\MOUSE.SYS

Driver	Type	Nom
0275:0048	C	NUL
0A88:0000	C	MS$MOUSE (MOUSE.SYS)
0A25:0000	C	CON (en fait, ANSI.SYS)
09C3:0000	C	EMMXXXX0 (QUEMM.SYS)
0070:016E	C	CON (pilote CON standard)
...	...	...
0070:0236	C	COM4

Tableau 3.8
Drivers en RAM.

Le programme MapDrv.Pas utilise la méthode décrite ci-dessus pour afficher l'emplacement mémoire, l'offset de la routine de stratégie, celui de la routine d'interruption, le type et le nom éventuel des différents drivers chargés en RAM.

Listing 3.9
Programme MapDrv.Pas. ❶

```
PROGRAM ListePilotesDePeripheriques; { MapDrv.Pas }

USES Dos, Crt, Sys;

TYPE
  DrvPtr  = ^Drivers;
  Drivers = RECORD
              Segment, Offset : Word;
              Suivant         : DrvPtr;
            END;
VAR  Liste   : DrvPtr;
     SegNUL,
     OfsNUL  : Word;
```

➥

Programme MapDrv.Pas (suite). ❷

```
FUNCTION SegDriverCON : Word;
VAR Regs : Registers;
BEGIN
  Regs.Ah:=$52;
  MsDos(Regs);
  IF Regs.Flags AND 1 = 1 THEN
    SegDriverCON := $FFFF
  ELSE
    SegDriverCON := MemW[Regs.Es:Regs.Bx+$E];
END;

FUNCTION OfsDriverCON : Word;
VAR Regs : Registers;
BEGIN
  Regs.Ah := $52;

  MsDos(Regs);
  IF Regs.Flags AND 1 = 1 THEN
    OfsDriverCon := $FFFF
  ELSE
    OfsDriverCON := MemW[Regs.Es:Regs.Bx+$C];
END;

FUNCTION SegDriverNUL : Word;
VAR Regs : Registers;
BEGIN
  Regs.Ah := $52;
  MsDos(Regs);
  IF (Regs.Flags AND 1 = 1) THEN
    SegDriverNUL := $FFFF
  ELSE
  BEGIN
    IF ((MemW[Regs.Es:Regs.Bx+$26] AND 4) SHR 2 = 1) THEN
      SegDriverNUL := Regs.Es
    ELSE
      SegDriverNUL := $FFFF;
  END;
END;

FUNCTION OfsDriverNUL : Word;
VAR Regs : Registers;
BEGIN
  Regs.Ah := $52;
  MsDos(Regs);
```

➡

Programme MapDrv.Pas (suite). ❸

```
  IF (Regs.Flags AND 1 = 1) THEN
    OfsDriverNUL := $FFFF
  ELSE
  BEGIN
    IF ((MemW[Regs.Es:Regs.Bx+$26] AND 4) SHR 2 = 1) THEN
      OfsDriverNUL := Regs.Bx + $22
    ELSE
      OfsDriverNUL := $FFFF;
  END;
END;

PROCEDURE Construit(VAR PtrDrv : DrvPtr);
VAR Actuel, Suivant : DrvPtr;
BEGIN
  Actuel := PtrDrv;
  WHILE Actuel^.Offset <> $FFFF DO
  BEGIN
    New(Suivant); Suivant^.Suivant := NIL;
    Suivant^.Segment:=MemW[Actuel^.Segment:Actuel^.Offset+2];
    Suivant^.Offset :=  MemW[Actuel^.Segment:Actuel^.Offset];
    Actuel^.Suivant := Suivant; Actuel := Suivant;
  END;
  Actuel := PtrDrv;
END;

PROCEDURE AfficheDrivers(PtrDrv : DrvPtr);
VAR Actuel, Suivant : DrvPtr;
    i, j            : Integer;
BEGIN
  Actuel := PtrDrv; Suivant := Actuel^.Suivant;
  i := 0; ClrScr;
  WHILE (Actuel^.Offset <> $FFFF) DO
  BEGIN
    IF WhereY >= 24 THEN
    BEGIN
      Write(' ------ Appuyez sur <┘ ------- ');
      ReadLn; ClrScr;
    END;
    Write(' ', MotDecVersHex(Actuel^.Segment), ':',
      MotDecVersHex(Actuel^.Offset) );
    WriteLn('':5, ' Ofs Routine de StratÄgie en ',
      MotDecVersHex(MemW[Actuel^.Segment:Actuel^.Offset+6]) );
    WriteLn('':15,'Ofs Routine d''Interruption en ',
      MotDecVersHex(
    MemW[Actuel^.Segment:Actuel^.Offset+8]) );
```

➥

Programme MapInt.Pas (suite). ❹

```
    Write('':25,'TYPE ', Chr(66 + (MemW[Actuel^.Segment:
    Actuel^.Offset+4] AND $8000 SHR 15)), ' ' );
    WITH Actuel^ DO
      IF (MemW[Segment:Offset+4] AND $8000 SHR 15 = 0 THEN
        Write(Mem[Actuel^.Segment:Actuel^.Offset+$A],
        ' Unités logiques gérées ')
      ELSE
        FOR j:=0 TO 7 DO
          Write(Chr (Mem[Actuel^.Segment:Actuel^.Offset+$A+j]));
    Actuel := Suivant; Suivant := Suivant^.Suivant;
    WriteLn; Inc(i);
  END;
  WriteLn;
  WriteLn('':20, i:2, ' Drivers de périphériques en RAM');
END;

BEGIN
  New(Liste); Liste^.Suivant := NIL;
  SegNUL := SegDriverNUL;
  OfsNUL := OfsDriverNUL;
  IF (SegNUL <> $FFFF) THEN
    Liste^.Segment := SegNUL;
  IF (OfsNUL <> $FFFF) THEN
    Liste^.Offset := OfsNUL;
  Construit(Liste);
  AfficheDrivers(Liste);
  Dispose(Liste); Liste := NIL;
END.
```

Blocs de paramètres disque

A la suite de l'initialisation des device drivers, le DOS construit un bloc de paramètres disque par lecteur logique (dont un pour `B:`, même s'il est absent, ce qui permet de copier une disquette de `A:` vers `B:` en n'ayant qu'un seul lecteur). Ces blocs de paramètres sont également chargés en RAM dans une liste chaînée. Outre un pointeur sur le prochain bloc de paramètres disque, ils contiennent quatre types de renseignements différents : identification par numéro d'unité et de sous-unité (pour les différencier lors d'un appel au driver, qu'ils partagent généralement), BIOS Parameter Block (aussi utilisé dans le driver d'unités de disque et dans le secteur boot du disque), pointeur sur le device driver correspondant et drapeau d'utilisation (`FFh` si le disque en question n'a pas encore été utilisé).

Adresse	***Signification***	***Type***
00h	Numéro du lecteur	octet
01h	Numéro de sous-unité	octet
02h	Octets par secteur	mot
04h	Secteurs - 1 par cluster	octet
05h	Décalage secteur ⟶ cluster	octet
06h	Offset vers la FAT	mot
08h	Nombre de FAT	octet
09h	Nombre d'entrées dans la racine	mot
0Bh	Offset vers les données	mot
0Dh	Dernier cluster de données	mot
0Fh	Secteurs par FAT	octet
10h	Offset vers la racine	mot
12h	Adresse de l'en-tête du device driver disque	2 mots (ofs : seg)
16h	ID Média	octet
17h	Drapeau d'utilisation	octet
18h	Prochain DBP	2 mots (ofs : seg)
1Ch	Inconnu	mot
1Eh	Inconnu	mot

Figure 3.10
Format d'un bloc de paramètres disque.

Parmi les nombreux renseignements fournis par le DBP, on remarquera les offset vers la FAT, la racine et les données, ainsi que deux masques (en `04h` et `05h`). Le premier n'est autre que le nombre de secteurs par cluster - 1 : il permet de trouver le numéro du dernier secteur d'un cluster donné. Le second permet de passer d'un numéro de secteur à un numéro de cluster : on applique pour cela la formule `"Cluster := Secteur SHR Valeur"`. On aura également noté le champ `0Dh`, qui donne le dernier cluster de données du disque, ainsi que les deux derniers champs, dont la signification reste à découvrir.

Le DOS alloue un minimum de deux blocs de paramètres disque au lancement du système. Chacun d'eux pointe sur le device driver standard de gestion des disques. L'adresse du premier bloc de paramètres disque est fournie par le nœud d'information du DOS (Int `21h`, fonction `52h`).

Le programme `MapDBP.Pas` construit la liste des blocs de paramètres disques présents en RAM, et affiche le contenu de quelques uns de leurs champs (adresse en mémoire, adresse du driver correspondant, numéro d'unité et de sous-unité, nombre d'entrées dans le répertoire racine).

Listing 3.11
Programme MapDBP.Pas. ❶

```
PROGRAM BlocsDeParametresDisques; {MapDBP.Pas}

USES Dos, Crt, Sys;

TYPE PtrDiskParam   = ^DiskParamBlock;
     DiskParamBlock = RECORD
                        Segment, Offset : Word;
                        Suivant         : PtrDiskParam;
                      END;

VAR Liste : PtrDiskParam;

FUNCTION SegPremierDBP : Word;
VAR Regs : Registers;
BEGIN
  Regs.Ah:=$52;
  MsDos(Regs);
  IF (Regs.Flags AND 1 = 1) THEN
    SegPremierDBP := $FFFF
  ELSE
    SegPremierDBP := MemW[Regs.Es:(Regs.Bx+2)];
END;

FUNCTION OfsPremierDBP : Word;
VAR Regs : Registers;
BEGIN
  Regs.Ah:=$52;
  MsDos(Regs);
  IF (Regs.Flags AND 1 = 1) THEN
    OfsPremierDBP := $FFFF
  ELSE
    OfsPremierDBP := MemW[Regs.Es:Regs.Bx];
END;

PROCEDURE ConstruitListe(VAR PtrListe : PtrDiskParam);
VAR  Actuel, Suivant : PtrDiskParam;
BEGIN
  Actuel := PtrListe;
  WHILE Actuel^.Offset <> $FFFF DO
```

➥

Programme MapDBP.Pas (suite). ❷

```
  BEGIN
    New(Suivant); Suivant^.Suivant := NIL;
    Suivant^.Segment := MemW[Actuel^.Segment:Actuel^.Offset+$1A];
    Suivant^.Offset  := MemW[Actuel^.Segment:Actuel^.Offset+$18];
    Actuel^.Suivant  := Suivant; Actuel := Actuel^.Suivant;
  END;
  Actuel := PtrListe;
END;

PROCEDURE ListeDBP(PtrListe : PtrDiskParam);
VAR i              : Integer;
    Actuel, Suivant : PtrDiskParam;
BEGIN
  Actuel := PtrListe; Suivant := Actuel^.Suivant;
  i:=0; ClrScr; WriteLn; WriteLn;
  WHILE (Actuel^.Offset <> $FFFF) DO
  BEGIN
    Write(' ',MotDecVersHex(Actuel^.Segment), ':',
          MotDecVersHex(Actuel^.Offset));
    Write('    ', Chr(65 +
          Mem[Actuel^.Segment:Actuel^.Offset]), ':');
    Write(', Sous-unité n° ',
          Mem[Actuel^.Segment:Actuel^.Offset+1]);
    Write('   utilisé : ');
    IF Mem[Actuel^.Segment:Actuel^.Offset+$17] = $FF THEN
      WriteLn('NON')
    ELSE
      WriteLn('OUI');
    Write('':10,' Secteur par Cluster : ',
          Mem[Actuel^.Segment:Actuel^.Offset+4]+1);
    Write('':13, MemW[Actuel^.Segment:Actuel^.Offset+9]);
    WriteLn(' entrées dans la racine');
    Write('':10,' En-Tête de Driver : ', MotDecVersHex
      (MemW[Actuel^.Segment:Actuel^.Offset+$14]) , ':');
    Write(MotDecVersHex
      (MemW[Actuel^.Segment:Actuel^.Offset+$12]));
    Write('':3, 'Prochain DBP : ', MotDecVersHex
      (MemW[Actuel^.Segment:Actuel^.Offset+$1A]) , ':');
    WriteLn(MotDecVersHex
      (MemW[Actuel^.Segment:Actuel^.Offset+$18]) );
    Actuel := Suivant; Suivant := Suivant^.Suivant; Inc(i);
    WriteLn;
  END;
  WriteLn('':20,i:2, ' Blocs de Paramètres Disque en RAM');
END;
```

➥

Programme MapDBP.Pas (suite). ❸

```
BEGIN
  New(Liste); Liste^.Suivant := NIL;
  IF (SegPremierDBP <> $FFFF) THEN
    Liste^.Segment := SegPremierDBP;
  IF (OfsPremierDBP <> $FFFF) THEN
    Liste^.Offset := OfsPremierDBP;
  ConstruitListe(Liste);
  ListeDBP(Liste);
  Dispose(Liste); Liste := NIL;
END.
```

Table des chemins

C'est sans doute juste après avoir créé les différents blocs de paramètres disque que le DOS initialise la table des chemins. On ne peut toutefois en être absolument sûr, Microsoft ne fournissant aucune indication à ce sujet. Comme son nom l'indique, la table des chemins contient le chemin courant pour chacun des disques logiques présents. La longueur de chaque enregistrement est de 80 octets. Les 64 premiers contiennent le chemin courant. Ensuite vient le mot d'état du périphérique, qui indique si le disque est en réseau, s'il est local, "joint" à un autre, ou "substitué" par le fait des commandes DOS : JOIN et SUBS. La fin de l'enregistrement contient un pointeur sur le bloc de paramètres disques correspondant à l'unité logique ainsi qu'un champ inconnu de huit octets.

Adresse	***Signification***	***Taille***
00h	Chemin en cours	64 octets
43h	Mot d'état	1 mot
45h	Adresse du DBP	2 mots (ofs : seg)
49h	Inconnu	8 octets

Une entrée de la table des chemins

Format du mot d'état du périphérique

Valeur	***Signification***
8000h	Périphérique réseau
4000h	Périphérique local
2000h	Périphérique joint
1000h	Périphérique substitué

Figure 3.12
Format de la table des chemins.

Le programme `MapPath.Pas` affiche les valeurs de chaque entrée de la table des chemins.

Listing 3.13
Programme MapPath.Pas. ❶

```
PROGRAM TrouveTableDesChemins; { MapPath.Pas }

USES Dos, Crt, Sys;

VAR
  Segment, Offset : Word;

FUNCTION DskActuel : Byte;
VAR Regs : Registers;
BEGIN
  Regs.Ah := $19;
  MsDos(Regs);
  IF Regs.Flags AND 1 = 1 THEN
    DskActuel := $F
  ELSE
    DskActuel := Regs.Al;
END;

FUNCTION NbLecteurs(DskCourant : Byte) : Byte;
VAR Regs : Registers;
BEGIN
  Regs.Ah:=$0E;
  Regs.Dl:=DskCourant;
  MsDos(Regs);
  IF Regs.Flags AND 1 = 1 THEN
    NbLecteurs := 0
  ELSE
    NbLecteurs := Regs.Al;
END;

FUNCTION SegTableChemins : Word;
VAR  Regs : Registers;
BEGIN
  Regs.Ah:=$52;
  MsDos(Regs);
  IF Regs.Flags AND 1 = 1 THEN
    SegTableChemins := $FFFF
  ELSE
    SegTableChemins := MemW[Regs.Es:Regs.Bx+$18];
END;
```

➥

Programme MapPath.Pas (suite). ❷

```
FUNCTION OfsTableChemins : Word;
VAR Regs : Registers;
BEGIN
  Regs.Ah := $52;
  MsDos(Regs);
  IF Regs.Flags AND 1 = 1 THEN
    OfsTableChemins := $FFFF
  ELSE
    OfsTableChemins := MemW[Regs.Es:Regs.Bx+$16];
END;

PROCEDURE AfficheChemins(NbLecteurs : Byte);
VAR  Chaine : PathStr;
     AdrSeg,
     AdrOfs,
     Etat   : Word;
     i,j    : Integer;
BEGIN
  Segment := SegTableChemins;
  Offset := OfsTableChemins;
  i:=0; ClrScr;
  WriteLn('':20, 'Table des chemins en ',
    MotDecVersHex(Segment),':',MotDecVersHex(Offset));
  IF (Segment <> $FFFF) AND (Offset <> $FFFF) THEN
    WHILE (i <= (NbLecteurs * $50)) DO
    BEGIN
      j:=0; Chaine:='';
      REPEAT
        Chaine := Chaine + Chr(Mem[Segment:Offset+i+j]);
        Inc(j);
      UNTIL (j = $43) Or (Mem[Segment:Offset+i+j] = 0);
      AdrSeg := MemW[Segment:Offset+$47];
      AdrOfs := MemW[Segment:Offset+$45];
      Etat   := MemW[Segment:Offset+$43];
      WriteLn('Chemin  : ', Chaine);
      WriteLn('Etat    : ', MotDecVersHex(Etat));
      WriteLn('Adresse : ', MotDecVersHex(AdrSeg),
       ':',MotDecVersHex(AdrOfs));
      Inc(i, $51);
    END
  ELSE
    Write('Erreur DOS (FUNCTION 52h, Int 21h)');
END;

BEGIN
  AfficheChemins(NbLecteurs(DskActuel));
END.
```

Buffers disques

L'étape suivante d'initialisation des structures DOS est représentée par l'allocation d'un buffer disque. Ce n'est qu'à la lecture du fichier CONFIG.SYS que le système allouera les suivants en fonction des besoins définis par l'utilisateur.

Adresse	*Signification*	*Taille*
00h	Prochain en-tête	2 mots (ofs : seg)
04h	Numéro du disque	octet
05h	Octet d'état	octet
06h	Numéro de secteur	mot
08h	Inconnu	mot
0Ah	DBP du disque	2 mots (ofs : seg)
0Eh	Drapeaux	mot
10h	Données	512 octets

Format du mot d'état du périphérique

Valeur	*Signification*
02h	Buffer = FAT
04h	Buffer = répertoire
08h	Buffer = données
20h	Buffer OK
40h	Buffer endommagé

Remarque : l'octet d'état peut avoir une valeur résultant de la combinaison d'une des trois premières (2h, 4h, 8h) avec l'une des deux dernières (20h, 40h). Ainsi, une valeur de 22h signifie que le buffer contient un secteur de la FAT et que ses données sont encore d'actualité et en bon état.

Figure 3.14
Format d'un en-tête de buffer disque.

Outre le pointeur sur le prochain buffer et le contenu du secteur lui-même, les champs intéressants d'un en-tête de buffer disque sont ceux qui concernent l'unité de disque, le numéro de secteur, le type du buffer (FAT, répertoire, ou données), et son état (endommagé ou lisible).

Les programmes utilitaires comme PCCACHE.COM (dans *PcTools*) allouent un buffer de plus dans la liste du DOS : on peut donc trouver plus de buffers que n'en déclare le fichier CONFIG.SYS en exécutant le programme MapBuf.Pas. Celui-ci fonctionne comme les précédents : il crée une liste de pointeurs contenant l'adresse de chaque en-tête de buffer et l'affiche en donnant tous les renseignements nécessaires.

Listing 3.15
Programme MapBuf.Pas.

❶

```
PROGRAM TrouveBuffers; { MapBuf.Pas }

USES Dos, Crt, Sys;

TYPE  BufPtr  = ^Buffer;
      Buffer  = RECORD
                  Segment, Offset : Word;
                  Suivant         : BufPtr;
                END;
VAR
  Liste : BufPtr;
FUNCTION SegPremierBuffer : Word;
VAR Regs : Registers;
BEGIN
  Regs.Ah:=$52;
  MsDos(Regs);
  IF (Regs.Flags AND 1 = 1) THEN
    SegPremierBuffer:=$FFFF
  ELSE
    SegPremierBuffer := MemW[Regs.Es:Regs.Bx+$14];
END;

FUNCTION OfsPremierBuffer : Word;
VAR Regs : Registers;
BEGIN
  Regs.Ah := $52;
  MsDos(Regs);
  IF (Regs.Flags AND 1 = 1) THEN
    OfsPremierBuffer := $FFFF
  ELSE
    OfsPremierBuffer := MemW[Regs.Es:Regs.Bx+$12];
END;

PROCEDURE Construit(VAR PtrBuf : BufPtr);
VAR Actuel, Suivant : BufPtr;
BEGIN
  Actuel:=PtrBuf;
  WHILE (Actuel^.Offset <> $FFFF) DO
  BEGIN
    New(Suivant); Suivant^.Suivant := NIL;
    Suivant^.Segment := MemW[Actuel^.Segment:Actuel^.Offset+2];
    Suivant^.Offset :=MemW[Actuel^.Segment:Actuel^.Offset];
    Actuel^.Suivant := Suivant; Actuel := Actuel^.Suivant;
  END;
```

➥

Programme MapBuf.Pas (suite) ❷

```
  Actuel := PtrBuf;
END;

FUNCTION BufInfos(Mot : Byte) : STRING;
VAR Reponse : STRING;
BEGIN
  Reponse := '';
  IF (Mot OR 2 = Mot) THEN
    Reponse := Reponse + ' FAT ';
  IF (Mot OR 4 = Mot) THEN
    Reponse := Reponse + ' Répertoire ';
  IF (Mot OR 8 = Mot) THEN
    Reponse := Reponse + ' Données ';
  IF (Mot OR $20 = Mot) THEN
    Reponse := Reponse + ' : Ok ';
  IF (Mot OR $40 = Mot) THEN
    Reponse := Reponse + ' : Endommagé ';
  BufInfos := Reponse;
END;

PROCEDURE AfficheBuffers(PtrBuf : BufPtr);
VAR Actuel, Suivant : BufPtr;
    i               : Integer;
BEGIN
  Actuel := PtrBuf; Suivant:=Actuel^.Suivant;
  i := 0; ClrScr;
  WHILE (Actuel^.Offset <> $FFFF) DO
  BEGIN
    IF (WhereY >= 24) THEN
    BEGIN
      Write(' ----- Appuyez sur ↵  ------- ');
      ReadLn; ClrScr;
    END;
    WriteLn;
    Write(' Secteur n° ',
         MemW[Actuel^.Segment:Actuel^.Offset+6]);
    Write(' de l''unité ', Chr( 65 +
          Mem[Actuel^.Segment:Actuel^.Offset+4]), ': ');
    Write('en ', MotDecVersHex(Actuel^.Segment), ':',
          MotDecVersHex(Actuel^.Offset+$10) );
    WriteLn(' (DBP en ',
            MotDecVersHex( MemW[Actuel^.Segment:
            Actuel^.Offset+$C]),':',
```

➡

Programme MapBuf.Pas (suite) ❸

```
    MotDecVersHex( MemW[Actuel^.Segment:
            Actuel^.Offset+$A]), ') ');
    WriteLn('':12, ' Type du Buffer : ',
         BufInfos(Mem[Actuel^.Segment: Actuel^.Offset+5]));
    WriteLn('':12,' En-Tête du Buffer en ',
            MotDecVersHex(Actuel^.Segment), ':',
            MotDecVersHex(Actuel^.Offset));
    WriteLn(' Nombre = ',
    MemW[Actuel^.Segment:Actuel^.Offset+8]);
    Actuel:=Suivant; Suivant:=Suivant^.Suivant; Inc(i);
  END;
  WriteLn; WriteLn('':25,i:2, '  Buffers en RAM');
END;

BEGIN
  New(Liste); Liste^.Suivant := NIL;
  IF SegPremierBuffer <> $FFFF THEN
    Liste^.Segment := SegPremierBuffer;
  IF OfsPremierBuffer <> $FFFF THEN
    Liste^.Offset := OfsPremierBuffer;
  Construit(Liste);
  AfficheBuffers(Liste);
  Dispose(Liste); Liste := NIL;
END.
```

Mémoire utilisateur

Toutes les structures de données dont nous avons parlé juqu'à présent restent cachées à l'utilisateur. Même s'il connaît l'existence des buffers et des drivers de périphériques, il n'a aucun moyen de savoir comment ils fonctionnent. En revanche, chacun est quotidiennement confronté aux problèmes de taille mémoire et de lancement de programmes par l'intermédiaire d'autres (qui n'a jamais exécuté la séquence de touches "<Esc>, B, D" sous Word, ou l'option "`File\Os Shell`" de Turbo Pascal ?). DOS met en œuvre plusieurs structures de données pour gérer le lancement de programmes et l'attribution de blocs de mémoire, dont la plus connue est le PSP. Mais il ne joue en fait qu'un rôle secondaire comparé à celui des MCB (blocs de contrôle de la mémoire) ou de la fonction EXEC.

Fonction EXEC (Int 21h, Fonction 4Bh)

Nous étudions la fonction EXEC avant les structures de données qu'elle met en œuvre parce que le DOS passe par elle lorsqu'il charge un programme en mémoire, et qu'en comprenant comment elle fonctionne, on saisira mieux l'utilité des MCB et du PSP.

EXEC n'est autre que le chargeur fourni par le DOS. Cette fonction a pour rôle de mettre un programme, ses données et sa pile en mémoire, de lui fournir le plus de renseignements possible sur la configuration du PC, de lui passer le contrôle (c'est-à-dire de l'exécuter), et – une fois le programme terminé – de rendre le contrôle de la machine à l'appelant (le plus souvent COMMAND.COM). Si de nombreux logiciels permettent de les quitter momentanément pour travailler avec le DOS, c'est également grâce à la fonction EXEC : en effet, rien n'interdit que le programme fils soit COMMAND.COM lui-même.

Lorsqu'on appelle la fonction EXEC, celle-ci commence par déterminer si elle doit charger un programme .EXE ou .COM en inspectant sa signature. S'il s'agit d'un fichier .EXE, son en-tête indique les tailles nécessaires à son bon fonctionnement. Si c'est un fichier .COM, on suppose que sa taille est exactement le minimum dont il a besoin pour fonctionner en mémoire. Dès lors, EXEC sait de combien de mémoire elle a besoin pour charger le fichier spécifié. Elle tente donc d'allouer deux blocs de mémoire, dont l'un contiendra le bloc d'environnement tandis que l'autre sera réservé au programme lui-même (avec son PSP, ses données, son code et sa pile). Tandis que les programmes .COM ont besoin de toute la mémoire disponible, les programmes .EXE définissent leurs besoins en mémoire par l'intermédiaire de deux champs de leur en-tête (voir chapitre 9, *Les fichiers .EXE*).

EXEC copie ensuite le bloc d'environnement du programme père dans le bloc de mémoire alloué à cette fin au programme fils, contruit un PSP au début du segment de mémoire qui contiendra le programme fils, et copie dans ce PSP la ligne de commande et les deux éventuels FCB que lui passe le programme père. Les valeurs des vecteurs d'interruptions 22h, 23h et 24h (terminer le programme, Ctrl-C et Erreur critique) sont sauvegardées dans trois champs du PSP ; le vecteur 22h est ensuite mis à jour de telle façon que le contrôle revienne au programme père si le programme fils se termine ou est interrompu.

Le code et les données du programme fils sont ensuite lus à partir du fichier présent sur le disque et chargés en mémoire juste au-dessus du nouveau PSP. Si le programme fils est un fichier .EXE, la table de relogement que contient son en-tête est utilisée pour mettre à jour les références aux segments que contient le code du programme en fonction de son adresse de chargement en mémoire.

Enfin, EXEC initialise les registres du CPU et la pile conformément au type du programme, qui prend ensuite le contrôle à son point d'entrée (0100h pour un fichier .COM, l'adresse spécifiée par son en-tête dans un fichier .EXE).

Il suffit de lire attentivement cette description sommaire des principes auxquels obéit la fonction EXEC pour comprendre qu'elle ne saurait faire son travail s'il n'y avait pas assez de mémoire disponible. D'autre part, il serait impensable de tenter d'utiliser cette fonction sans connaître le format du PSP. Il nous faut donc étudier d'une part les fonctions DOS d'attribution mémoire et les MCB, d'autre part les PSP. Avant de nous engager dans cette voie, le tableau qui suit récapitule les différences entre l'appel de la fonction EXEC par l'intermédiaire de l'assembleur et la procédure Turbo Pascal de l'unité DOS EXEC.

Fonction EXEC en Pascal et en Assembleur

1. Avec Turbo Pascal

```
PROGRAM ExecuteFils;  {ExecPrg.Pas}
{ Fixer la Taille Max du Tas au minimum, pour que les  }
{ programmes fils puissent s'exécuter sans difficulté  }
{$M 16384, 0, 2000}
USES Dos;
PROCEDURE Erreur;
BEGIN
  IF (DosError <> 0) THEN
    WriteLn(' Erreur n° ', DosError)
  ELSE
    WriteLn(' Code de Sortie n° ', DosExitCode);
END;
PROCEDURE Directory(Arguments : STRING);
BEGIN
  WriteLn('Dir sous Turbo-Pascal');
  SwapVectors;                       { Obligatoire }
  { COMMAND.COM /C permet de décharger la copie de     }
  { l'interpréteur de commandes après l'avoir appelé  }
  Exec(GetEnv('COMSPEC'), Concat('/C Dir ',Arguments));
  SwapVectors;                       { Obligatoire }
END;

PROCEDURE PcTools;
BEGIN
  WriteLn(' Directory par PcTools ');
  SwapVectors;                       { Obligatoire }
  Exec('C:\PCTOOLS\PCSHELL.EXE', '');
  SwapVectors;                       { Obligatoire }
End;
```

```
Begin
  Directory('D:\Pascal\Sources\*.Pas /p /w');
  Erreur;
  PcTools;
  Erreur;
  ReadLn;
End.
```

2. Avec l'Assembleur : paramètres à passer (pas d'exemple)

Numéro de fonction	*Paramètres*	*Retour*
4Bh	AH := 4Bh	Si CF=1, Ax = Erreur
	AL := 00h	
	:= 03h (Overlay)	
	ES := Seg(BlocParam)	
	BX := Ofs(BlocParam)	
	DS := Seg(NomChemin ASCIIZ)	
	DX := Ofs(NomChemin ASCIIZ)	

Le bloc de paramètres dont l'adresse doit être passée en ES : BX a la structure suivante (pour la sous-fonction 00h) :

Déplacement	*Description*
00h	Segment du bloc d'environnement
02h	Offset de la ligne de commande
04h	Segment de la ligne de commande
06h	Offset du 1° FCB copié dans nouveau PSP
08h	Segment du 1° FCB copié dans nouveau PSP
0Ah	Offset du 2° FCB copié dans nouveau PSP
0Ch	Segment du 2° FCB copié dans nouveau PSP

Remarque — L'appel de la fonction Exec par l'intermédiaire de Turbo Pascal a sur l'Assembleur (ou sur un appel effectué avec le type Registers de Turbo Pascal) l'avantage de la simplicité : ni le bloc d'environnement ni la ligne de commande n'ont besoin d'être gérés par le programmeur. En revanche, de nombreux essais sont nécessaires avant de déterminer une bonne taille pour le Tas, que l'on doit indiquer au compilateur avec la directive {$M}.

Fonctions DOS d'attribution de mémoire

L'Int 21h du DOS dispose de trois fonctions destinées à la gestion de la mémoire RAM. Ces trois fonctions (numéros 48h, 49h et 4Ah) permettent d'allouer et de désallouer des blocs de mémoire, ou de modifier la taille d'un bloc de mémoire déjà alloué. Ce sont celles-là qu'EXEC utilise pour charger un programme. On note enfin une dernière fonction (n° 58h), qui permet de sélectionner l'algorithme de recherche des blocs mémoire employé par le DOS.

Fonction numéro	*Paramètres*	*Retour*
48h	AH := 48h	AX= Segment alloué
	BX := NbParag ou Code d'Erreur	BX= Taille en paragraphes du plus grand bloc disponible (si erreur)
49h	AH := 49h	AX=Code d'erreur
	ES := Segment à libérer	
4Ah	AH := 4Ah	AX=Code d'erreur
	BX := NbParag	BX=Taille en Paragraphes du plus grand bloc
	ES := Segment à modifier	
58h	AH := 58h	AX = Algorithme si lecture et CF=0
	AL := 00h (Lecture)	
	:= 01h (Ecriture)	= Code d'Erreur si écriture et CF=1
	BX := Algorithme	

Les codes d'algorithme pour la fonction 58h sont les suivants :

Code	*Algorithme*
00h	Premier bloc disponible (par défaut)
01h	Meilleur bloc (le plus petit suffisant à l'allocation)
02h	Dernier bloc disponible

Tableau 3.16
Fonctions DOS d'attribution de mémoire.

Le Turbo Pascal utilise ces fonctions lors de la création et la destruction de pointeurs par `New`, `GetMem`, `Mark`, `Dispose`, `FreeMem` et `Release`, et dans les fonctions prédéfinies `MemAvail` et `MaxAvail`. Le programmeur peut cependant (et doit, s'il souhaite utiliser `EXEC` dans son programme) y faire également appel, par exemple pour réduire la taille mémoire allouée à son programme. En effet, le DOS alloue par défaut toute la mémoire disponible à un programme. Lorsqu'il en est ainsi, il est évidemment impossible de lancer un programme fils.

Listing 3.17
Programme Memoire.Pas. ❶

```
PROGRAM Memoire;       { Memoire.Pas }
                       { Le Tas peut être encore plus grand }
{$M 16384, 0, 65536}

USES Dos;

VAR  SegPrg, Resultat : Word;
     MaxParag         : LongInt;

FUNCTION AlloueMemoire(NbPara : Word;
                       VAR Segment : Word) : Word;
VAR Regs : Registers;
BEGIN
 WITH Regs DO
 BEGIN
   Ah := $48;
   Bx := NbPara;
   MsDos(Regs);
   Segment := Ax;              { N° de Segment du Bloc }
   IF (Flags AND 1 = 1) THEN
     AlloueMemoire := Bx       { Maximum Disponible }
   ELSE
     AlloueMemoire := 0;
 END;
END;

FUNCTION RendMemoire(Segment : Word) : Word;
VAR Regs : Registers;
BEGIN
 WITH Regs DO
 BEGIN
   Ah := $49;
   Es := Segment;
   MsDos(Regs);
```

➥

Programme Memoire.Pas (suite) ❷

```
    IF (Flags AND 1 = 1)  THEN
      RendMemoire := Ax         { Erreur }
    ELSE
      RendMemoire := 0;
  END;
END;

FUNCTION ModifieMemoire(NbPara, Segment : Word) : Word;
VAR Regs : Registers;
BEGIN
 WITH Regs DO
 BEGIN
   Ah := $4A;
   Bx := NbPara;
   Es := Segment;
   MsDos(Regs);
   IF (Flags AND 1 = 1) THEN
     ModifieMemoire := Bx { Plus grand bloc Disponible }
   ELSE
     ModifieMemoire := 0;
 END;
END;

BEGIN
 Resultat := AlloueMemoire($FFFF, SegPrg);
 MaxParag := Resultat;
 MaxParag := (MaxParag SHL 4)
 WriteLn(' Mémoire disponible ',MaxParag Div 1024, '
           Ko');
 Resultat := AlloueMemoire($800, SegPrg);
 IF (Resultat = 0) THEN
   WriteLn('Allocation de 32 Ko OK ')
 ELSE
   WriteLn('Erreur, bloc le plus grand = ',Resultat);
 Resultat := ModifieMemoire($400, SegPrg);
 IF (Resultat <> 0) THEN
   WriteLn(' Erreur : ', Resultat)
 ELSE
   WriteLn(' Réduction à 16 Ko OK ');
 Resultat := RendMemoire(SegPrg);
 IF (Resultat <> 0) THEN
   WriteLn(' Erreur : ', Resultat)
 ELSE
   WriteLn(' Mémoire restante rendue au DOS ');
END.
```

MCB (blocs de contrôle de la mémoire)

Nous y faisions déjà allusion dans l'introduction à ce chapitre, nous en avons reparlé incidemment depuis, mais nous ne les avons pas encore étudiés : les MCB sont le nœud des fonctions DOS d'attribution de la mémoire. C'est grâce à cette structure de données que le système d'exploitation sait si un bloc de mémoire est constitué de données ou d'un programme exécutable. C'est aussi grâce à la souplesse de son fonctionnement qu'il est possible de programmer des applications dont la taille en mémoire est susceptible d'être modifiée. En bref, ils sont indispensables à la bonne marche du PC en général et de la RAM en particulier.

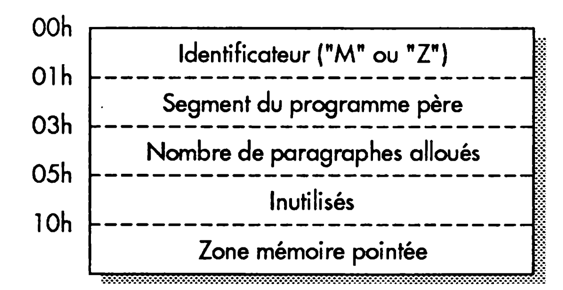


Figure 3.18
Format d'un MCB.

L'identificateur indique s'il s'agit ou non du dernier MCB. Le segment du programme père permet de déterminer par qui un programme pointé par un MCB a été chargé en mémoire et quelle est sa nature (zone libre, programme, données, environnement, PSP). La taille est exprimée en paragraphes : ajoutée à l'adresse du MCB actuel + 1, elle permet de trouver le suivant. Ensuite viennent onze octets dont on ne connaît pas la signification, et qui sont peut-être tout bonnement inutilisés par le DOS. Puis, la zone mémoire pointée par le MCB, qui se trouve en `SegMCB+1:0` (ou `SegMCB:OfsMCB+0Ah`).

L'adresse du premier MCB est donnée par la fonction `52h` de l'Int `21h` (voir plus haut). Lorsqu'on la connaît, on peut remonter la chaîne très facilement : il suffit d'ajouter à chaque fois la taille de la zone mémoire pointée à l'adresse du MCB + 1 pour avoir l'adresse du MCB suivant (pour les détails, voir le programme `MapMCB.Pas`). Les difficultés viennent ensuite : comment savoir s'il s'agit d'un programme, de données, d'une zone de mémoire libre, ou d'un environnement ? Commençons par le plus simple : distinguer programme et environnement ne pose pas de problème. Un bloc contenant un programme commence en fait par son PSP, et l'on sait que les deux premiers octets d'un PSP contiennent une instruction d'interruption `20h`. On lira donc le premier mot du bloc : si sa valeur est `20CDh`, on peut être assuré qu'il s'agit d'un PSP. Un bloc d'environnement, lui, débute toujours par la chaîne "`COMSPEC=`". Là encore, il suffit de vérifier.

Mais tout n'est pas si facile : de nombreux blocs mémoire ne sont pas identifiables par ces méthodes et appartiennent cependant à l'une des deux catégories précédentes. D'autres ne sont ni des programmes ni des blocs d'environnements : il peut s'agir de blocs libres, ou bien encore du programme `IBMDOS.COM` (qui n'a pas de PSP), ou de tampons, ou enfin de PSP détruits par leur programme. Il est alors un peu plus difficile de s'y retrouver. En ce qui concerne `IBMDOS.COM`, son propriétaire est déclaré comme ayant l'adresse de segment `0008h`. Les blocs libres, eux, ont un propriétaire de segment `0000h`. Quant aux PSP détruits, il est impossible de les identifier. En revanche, il est possible de reconnaître un bloc d'environnement qui ne commencerait par "`COMSPEC=`" : en effet, le champ numéro `2Ch` du PSP indique l'adresse du bloc d'environnement dont il dépend. On peut donc, par recoupements, remettre un bloc d'environnement au format bizarre à sa place.

Un autre problème apparaît dans la liste des programmes : lorsqu'on a lancé plusieurs programmes les uns à la suite des autres (grâce aux options de "Shell" que possèdent la plupart des logiciels actuels) et que l'on a une mémoire conventionnelle proche de la saturation, on voit apparaître dans la liste des programmes des blocs alloués de même taille (sur le PC de l'auteur, ils font 3 616 octets) entre deux programmes exécutables. C'est le chargeur du DOS (la partie transitoire de `COMMAND.COM`) qui a été appelé. Naturellement, il a aussi son PSP et son bloc d'environnement. Dans son PSP, on peut trouver au déplacement `5Ch` (1° FCB) le nom d'un fichier de données. Si, par exemple, vous lanciez le Turbo Pascal à partir de Word en lui passant un nom de fichier en ligne de commande, et que vous exécutiez `MapMCB.Pas`, vous verriez un bloc mémoire alloué d'à peu près 3 616 octets affichant le nom du programme que vous avez passé en ligne de commande au Turbo Pascal. Cela s'explique aisément : relisez le tableau des spécifications de la fonction `Exec` du Turbo Pascal, et vous verrez que sa syntaxe est : "`Exec(NomChemin\NomProgramme, Parametres)`". Lorsque Word appelle `Exec`, il lui passe aussi ces deux paramètres (au moins), ce qui donne `Exec('Command.Com /C D:\Pascal\Turbo', 'NomFic.Pas')`.

Le nom des autres programmes est encore un peu plus difficile à obtenir : c'est leur bloc d'environnement qui le contient dans la dernière de ses chaînes (à partir de la version 3.0 du DOS). Cette chaîne est séparée des autres par plusieurs zéros consécutifs. On ne peut donc afficher le nom du programme qu'après avoir été lire dans le PSP l'adresse du bloc d'environnement, et dans le bloc d'environnement toutes les chaînes jusqu'à la dernière...

Listing 3.19
Programme MapMCB.Pas. ❶

```
PROGRAM MapMCB;  { MapMCB.Pas }

USES Crt,Dos, Sys;

TYPE
 MCBPtr   =  ^MCB;
 MCB      =  RECORD
               Segment, Offset : Word;
               TypeMCB         : Char;
               Suivant         : MCBPtr;
             END;

VAR  Liste : MCBPtr;

FUNCTION SegPremierMCB : Word;
VAR Regs : Registers;
BEGIN
 WITH Regs DO
 BEGIN
   Ah := $52;
   MsDos(Regs);
   IF (Flags AND 1 = 1) THEN
     SegPremierMCB := $FFFF
   ELSE
     SegPremierMCB := MemW[Es:Bx-2];
 END;
END;

FUNCTION OfsPremierMCB : Word;
VAR Regs : Registers;
BEGIN
 WITH Regs DO
 BEGIN
   Ah := $52;
   MsDos(Regs);
   IF (Flags AND 1 = 1) THEN
     OfsPremierMCB := $FFFF
   ELSE
     OfsPremierMCB := MemW[Es:Bx-4];
 END;
END;
```

➥

Programme MapMCB.Pas (suite) ❷

```
PROCEDURE CreeListeMCB(VAR PtrMCB : MCBPtr);
VAR Actuel, Suivant : MCBPtr;
    ChaineEnv       : STRING[8];
BEGIN
 Actuel := PtrMCB; ChaineEnv := 'ZZZZZZZZ';
 REPEAT
   New(Suivant); Suivant^.Suivant := NIL;
   Suivant^.TypeMCB := #0;
   Suivant^.Segment := Actuel^.Segment + 1
                    + MemW[Actuel^.Segment:3];
   Suivant^.Offset := 0;
   IF (MemW[Suivant^.Segment + 1:0] = $20CD) THEN
     Suivant^.TypeMCB := 'P'      { PSP puis Programme }
   ELSE
   IF (Mem[Suivant^.Segment + 1 : 0] = Ord('C')) THEN
   BEGIN
     Move(Mem[Suivant^.Segment + 1 : 0],
          ChaineEnv[1], 8);
     IF (ChaineEnv = 'COMSPEC=') THEN
       Suivant^.TypeMCB := 'E';   { Environnement }
     ChaineEnv := 'ZZZZZZZZ';
   END
   ELSE
   IF ((Suivant^.TypeMCB = #0) AND
      (MemW[Suivant^.Segment : 1] = 0)) THEN
     Suivant^.TypeMCB := 'L';       { Libre }
   IF ((Actuel^.TypeMCB = #0) AND
      (MemW[Actuel^.Segment : 1] = 8)) THEN
     Actuel^.TypeMCB := 'P'         { Prog. IBMDOS.COM }
   ELSE
   IF ((Actuel^.TypeMCB = #0) AND
    (MemW[Suivant^.Segment+1:0] = $20CD) AND
    (MemW[Suivant^.Segment+1:$2C] = Actuel^.Segment+1))
     THEN
      Actuel^.TypeMCB := 'E'        { Env Anormal }
   ELSE
   IF (Actuel^.TypeMCB = #0) THEN
     Actuel^.TypeMCB := '?';        { Inconnu }
   Actuel^.Suivant := Suivant; Actuel := Suivant;
 UNTIL (Mem[Actuel^.Segment:0] = Ord('Z'));
 Actuel := PtrMCB;
END;
```

➡

Programme MapMCB.Pas (suite) ❸

```
PROCEDURE Ecran;
CONST Titre : STRING[11] = ' MCB en RAM ';
      Chaine: STRING[73] =
'Seg(MCB) | Parag. occupés | Taille     | Propr. | TYPE'+
'| Environnement ';

BEGIN
 TextAttr:=14+4*16;
 GotoXy(40-5, 1); Write(Titre);
 GotoXy(40-(72  DIV 2), 3); Write(Chaine);
END;

PROCEDURE AfficheMCB(PtrMCB : MCBPtr);
VAR Actuel          : MCBPtr;
    i, NbL, NbMCB,
    EnvSeg, TailEnv: Word;
    ChainesEnv      : ARRAY[0..1023] OF Byte;
    Chaine          : STRING;
    Taille          : LongInt;

   FUNCTION Verifie(Chaine : STRING) : Boolean;
   VAR i : Word;
   BEGIN
     REPEAT
       IF (Pos(' ', Chaine) <> 0) THEN
         Delete(Chaine, Pos(' ', Chaine), 1);
     UNTIL (Pos(' ', Chaine) = 0);
     Verifie := (Chaine <> '');
   END;

   PROCEDURE EcritParam(Adr : Word);
   CONST SetOk : SET OF Char = [#$20..'?','A'..'}'];
   VAR   Chaine: STRING;
         Ok    : Boolean;
   BEGIN
     Chaine := ''; TextAttr:=14+4*16;
     WHILE ((Mem[Actuel^.Segment + 1 : Adr+i] > 0) AND
       (Chr(Mem[Actuel^.Segment + 1:Adr + i]) IN SetOk)) DO
     BEGIN
       Chaine := Chaine +Chr(Mem[Actuel^.Segment+1 : Adr+i]);
       Inc(i);
     END;
     Ok := Verifie(Chaine);
     IF Ok THEN
```

➥

Programme MapMCB.Pas (suite) ❹

```
      BEGIN
        GotoXy(WhereX,NbL); Write(' ',Chaine,' ');
      END;
      TextAttr:=15+1*16;
    END;

BEGIN
 TextAttr := 15+1*16; ClrScr; Ecran;
 Actuel := PtrMCB; NbL := 5; NbMCB := 1; GotoXy(5,5);
 FillChar (ChainesEnv, SizeOf(ChainesEnv), #0);
 REPEAT
   IF (NbL >= 24) THEN
   BEGIN
     Write('--- Appuyez sur <Ø pour continuer ---');
     ReadLn; NbL := 5; ClrScr; Ecran;
   END;
   GotoXy(5, NbL); TextAttr := 15+1*16;
   Write( MotDecVersHex(Actuel^.Segment));
   GotoXy(17, NbL);
   Write( MotDecVersHex(Actuel^.Segment+1), '-');
   Write( MotDecVersHex(Actuel^.Segment+1+
          MemW[Actuel^.Segment:3]));
   Taille := MemW[Actuel^.Segment:3];
   Taille := Taille SHL 4;
   IF (Taille > MemAvail) THEN
     Taille := (Taille - MemAvail);
   GotoXy(31, NbL); Write(Taille:8);
   GotoXy(45, NbL);
   Write( MotDecVersHex(MemW[Actuel^.Segment:1]));
   GotoXy(56, NbL); Write( Actuel^.TypeMCB);
   IF (Actuel^.TypeMCB = 'P') THEN
   BEGIN
     EnvSeg := MemW[Actuel^.Segment+1:$2C];
     TailEnv:= MemW[EnvSeg-1 : 3] SHL 4;
     GotoXy(64, NBL); Write(MotDecVersHex(EnvSeg) );
     Inc(NbL);
     IF (MemW[Actuel^.Segment:1] = 8) THEN
     BEGIN
       GotoXy(5,NbL); TextAttr := 14+4*16;
       Write(' IbmBio.Com '); Inc(NbL);
     END
     ELSE
```

➥

Programme MapMCB.Pas (suite) ❺

```
      BEGIN
        FillChar (ChainesEnv, SizeOf(ChainesEnv), #0);
        Move(Mem[EnvSeg:0], ChainesEnv[0], TailEnv);
        i := 0;
        REPEAT
          Inc(i);
        UNTIL ((ChainesEnv[i]=0) AND (ChainesEnv[i+1]=0));
        Inc(i,4);
        IF (ChainesEnv[i] <> 0) THEN
        BEGIN                         { Nom du Prg dans Env }
          Chaine := '';
          WHILE (ChainesEnv[i] <> 0) DO
          BEGIN
            Chaine := Chaine+Chr(ChainesEnv[i]); Inc(i);
          END;
          GotoXy(5,NbL); TextAttr := 14+4*16;
          Write(' ',Chaine,' '); i:=1;
          EcritParam($80); Inc(NbL)    { Ligne de commande
                                         dans PSP }
        END
        ELSE
        BEGIN
          GotoXy(5,NbL); TextAttr:=14+4*16;
          Write(' Command.Com /C ');
          i:=1; EcritParam($5C);       { Lire FCB n°1 }
          Inc(NbL);
        END;
      END;
    END
    ELSE
    IF (Actuel^.TypeMCB = 'L') THEN
    BEGIN
      Inc (Nbl); GotoXy(5, NbL); TextAttr := 14+4*16;
      Write(' Bloc Libre ');Inc(Nbl);
    END;
    Inc(NbL); Inc(NbMCB); WriteLn;
    IF (Actuel^.Suivant <> NIL) THEN
      Actuel := Actuel^.Suivant;
  UNTIL (Mem[Actuel^.Segment:0] = Ord('Z'));
  Write; WriteLn(' ',NbMCB: 2, ' MCB en RAM');
  WriteLn(' Mémoire conventionnelle disponible ',
           MemAvail DIV 1024, ' Ko');
END;
```

➥

Programme MapMCB.Pas (suite) ❻

```
BEGIN
 New(Liste); Liste^.Suivant := NIL;
 Liste^.TypeMCB := #0;
 IF (SegPremierMCB <> $FFFF) THEN
   Liste^.Segment := SegPremierMCB;
 IF (OfsPremierMCB <> $FFFF) THEN
   Liste^.Offset := OfsPremierMCB;
 CreeListeMCB(Liste);
 AfficheMCB(Liste);
 Dispose(Liste); Liste := NIL;
END.
```

PSP (préfixe de segment de programme)

Il devrait être logique de parler des PSP après avoir examiné les MCB, puisqu'un PSP n'est créé en mémoire qu'à la condition qu'un MCB pointant sur un bloc mémoire suffisamment large pour le contenir soit alloué. En fait, on a vu dans la partie précédente qu'il est impossible de parler des MCB sans parler des PSP et inversement. C'est que les différentes structures DOS de gestion de la mémoire sont si liées entre elles qu'on ne peut travailler sur l'une en faisant l'économie de l'autre. Autrement dit, même si vous croyez déjà savoir ce qu'est un PSP, mieux vaut lire ce qui suit, qui apportera sans doute des précisions sur ce qui précède.

Nous avons dit plus haut (voir la fonction `EXEC`) que le DOS allouait automatiquement deux blocs de mémoire par programme qu'il chargeait. Le premier est réservé aux chaînes d'environnement, le second contient le PSP et le programme chargé. `EXEC` commence par appeler la fonction `26h` de l'Int `21h` pour construire le nouveau PSP. Cela fait, il initialise le champ `80h` du PSP en y copiant la ligne de commande à l'exception des opérateurs de redirection et des fichiers recevant la redirection. Le PSP est alors complètement à jour. Après avoir copié le bloc d'environnement à sa nouvelle adresse, le DOS charge enfin le programme à exécuter juste au dessus du PSP créé.

- Les champs `00h`, `05h`, `50h`, `55h`, `5Ch` et `6Ch` ne sont plus guère utilisés par les programmeurs depuis que `CP/M` a complètement cédé la place au DOS. En revanche, les autres fournissent de précieux renseignements.
- Le champ `02h` donne l'adresse de segment qui suit la fin du programme, ou – lorsqu'un seul programme est chargé en mémoire – l'adresse du dernier segment utilisable, généralement `A000h`.
- Le champ `04h` est inconnu : on peut supposer qu'il s'agissait du numéro d'ordre du PSP, qui serait finalement inutilisé.

00h	Int 20h	mot
02h	Dernier segment alloué	mot
04h	Inconnu - numéro du PSP ?	octet
05h	Appel FAR aux fonctions DOS	5 octets
0Ah	Adresse de l'Int 22h	2 mots (ofs : seg)
0Eh	Adresse de l'Int 23h	2 mots (ofs : seg)
12h	Adresse de l'Int 24h	2 mots (ofs : seg)
16h	Adresse de segment du PSP père	1 mot
18h	File handle table	20 octets
2Ch	Adresse de segment du bloc d'environnement	1 mot
2Eh	Pile du programme suivant le PSP	2 mots (SP : SS)
32h	Nombre de handles	1 mot
34h	Adresse de la file handle table	2 mots (ofs : seg)
38h	Adresse du prochain PSP. Inutilisé (ou adresse d'une seconde Fht ?)	2 mots (ofs : seg)
3Ch	Inconnu (ou seonde Fht ?)	20 octets
50h	Int 21h, RetF	3 octets
53h	Inconnu	1 mot
55h	Extension du premier FCB	7 octets
5Ch	FCB numéro 1	16 octets
6Ch	FCB numéro 2	16 octets
7Ch	Inconnu	2 mots
80h	DTA : ligne de commande	128 octets

Figure 3.20
Format du PSP.

- Les champs `0Ah`, `0Eh` et `12h` conservent les valeurs des vecteurs d'interruption `22h`, `23h` et `24h`. Ces vecteurs sont les seuls que le DOS suppose devoir être détournés par un programme. Il en conserve donc les dernières valeurs, de façon à ce que le programmeur puisse les restaurer sans difficultés, ce qui est d'un grand secours lorsqu'on écrit un résident.

- Le champ 16h donne l'adresse du PSP père : c'est généralement celle de COMMAND.COM.
- Le champ 18h contient la File handle table, qui mémorise les handles des différents fichiers utilisés dans un programme et permet la redirection d'un handle vers un autre (voir le chapitre 8, *Fichiers de données*).
- Le champ 2Ch fournit l'adresse du bloc d'environnement, qui contient entre autres le nom du programme suivant le PSP.
- Le champ 2Eh sert au stockage de la pile du programme lorsqu'il fait appel à des fonctions DOS qui utilisent la pile du système – c'est notamment le cas lorsqu'il passe provisoirement la main à COMMAND.COM ou lorsqu'il fait appel à la fonction EXEC.
- Le champ 32h indique le nombre de handles (20 par défaut) disponible pour le programme.
- Le champ 34h mémorise l'adresse de la File handle table, ce qui permet de disposer de deux tables : il suffit pour cela de sauvegarder la précédente adresse, puis de détourner ce champ vers la nouvelle table.
- Le champ 38h est inutilisé et devait être prévu pour contenir l'adresse du prochain PSP. Comme ce champ est un double mot, on ne peut s'empêcher de penser qu'il pourrait également contenir l'adresse d'une seconde File handle table (le DOS permet en effet de modifier le nombre de handles dont dispose un programme). D'autant que le champ suivant (3Ch), inconnu lui aussi, a une longueur de 20 octets – ce qui correspond bien à la taille par défaut d'une File handle table.
- Les champs 53h et 7Ch sont inconnus et initialisés à zéro.
- Le champ 80h, enfin, contient la ligne de commande : son premier octet donne la taille de la chaîne, qui commence donc à l'offset 81h.

Le joli petit programme qui suit, MapPSP.Pas, affiche les différents PSP qui se trouvent en mémoire, et permet à l'utilisateur de les examiner champ par champ. S'il vous paraît difficile à lire, vous pouvez tout à fait vous en dispenser, étant donné qu'il ne fait que reprendre les techniques que nous avons déjà vues dans ce chapitre. Les éléments nouveaux concernent essentiellement le déplacement du curseur et l'affichage à l'écran.

Listing 3.21
Programme MapPSP.Pas. ❶

```
PROGRAM MontreChampsPSP; { MapPSP.Pas }

USES Dos, Crt, Sys;

TYPE
  PtrPSP = ^PSP;
  PSP    = RECORD
             Segment, Offset : Word;
             Suivant         : PtrPSP;
           END;
VAR
  Liste : PtrPSP;
  NbPSP : Byte;
  Segment : Word;

FUNCTION PremierPSP : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $52;
    MsDos(Regs);
    IF Flags AND 1 = 1 THEN
      PremierPSP := $FFFF
    ELSE
      PremierPSP := MemW[Es:Bx-2];
  END;
END;

FUNCTION ListePSP(VAR PtrListe : PtrPSP) : Byte;
VAR Actuel, Suivant : PtrPSP;
BEGIN
  Actuel := PtrListe; NbPSP := 0;
  REPEAT
    IF (Actuel^.Segment+1) = MemW[Actuel^.Segment:1] THEN
    BEGIN
      Inc(NbPSP);
      New(Suivant); Suivant^.Suivant := NIL;
      Suivant^.Offset := 0;
      Suivant^.Segment := Actuel^.Segment + 1 +
        MemW[Actuel^.Segment:3];
      Actuel^.Suivant := Suivant; Actuel := Suivant;
    END
```

➥

Programme MapPSP.Pas (suite) ❷

```
    ELSE
      Actuel^.Segment := Actuel^.Segment + 1 +
        MemW[Actuel^.Segment:3];
  UNTIL Mem[Actuel^.Segment:0] = Ord('Z');
  Actuel^.Segment := $FFFF;
  ListePSP := NbPSP;
END;

PROCEDURE AfficheAdresses(PtrListe : PtrPSP);
VAR Actuel : PtrPSP;
    i      : Word;

    PROCEDURE AfficheNom;
    VAR EnvSeg, TailleEnv : Word;
        ChaineEnv         : ARRAY[0..1023] OF Byte;
        i                 : Word;

        PROCEDURE EcritParam(Adr : Word);
        CONST SetOk : SET OF Char = [#$20..'?', 'A'..'}'];
        VAR   i     : Word;
        BEGIN
          i:=1;
          WHILE (Mem[Actuel^.Segment+1:Adr+i] > 0) AND
            (Chr(Mem[Actuel^.Segment+1:Adr+i]) IN SetOk) DO
          BEGIN
            Write(Chr(Mem[Actuel^.Segment+1:Adr+i]) );
            Inc(i);
          END;
          WriteLn;
        END;

    BEGIN
      FillChar(ChaineEnv, SizeOf(ChaineEnv), #0);
      EnvSeg := MemW[Actuel^.Segment+1:$2C];
      TailleEnv := MemW[EnvSeg-1:3] SHL 4;
      Move(Mem[EnvSeg:0], ChaineEnv[0], TailleEnv);
      i:=0;
      REPEAT
        Inc(i);
      UNTIL (ChaineEnv[i] = 0) AND (ChaineEnv[i+1] = 0);
      Inc(i,4);
      IF (ChaineEnv[i] <> 0)  THEN
      BEGIN
        WHILE (ChaineEnv[i] <> 0) DO
```

➡

Programme MapPSP.Pas (suite) ❸

```
        BEGIN
          Write(Chr(ChaineEnv[i])); Inc(i);
        END;
        EcritParam($80);
      END
      ELSE
      BEGIN
        Write(' Command.Com /C ');
        EcritParam($5C);
      END;
    END;

BEGIN
  TextAttr:=14+4*16; GotoXy(34,1); Write(' PSP en RAM ');
  TextAttr:=15+5*16; GotoXy(1,3); Write('':80);
  GotoXy(1,3);
  Write(' Adresse MCB ≥  Adresse PSP ≥   '+'   Programme pointé');
  Actuel := PtrListe; i:=0;
  REPEAT
    GotoXy(2,5+i); Write('':78);
    GotoXy(15,5+i); Write('≥'); GotoXy(32,5+i); Write('≥');
    GotoXy(5,5+i); Write(MotDecVersHex(Actuel^.Segment));
    GotoXy(22, 5+i); Write(MotDecVersHex(Actuel^.Segment+1));
    GotoXy(37,5+i); AfficheNom; Inc(i);
    IF (Actuel^.Suivant <> NIL) THEN
        Actuel := Actuel^.Suivant;
  UNTIL (Actuel^.Suivant = NIL);
END;

PROCEDURE ChampsPSP(AdrPSP : Word);
VAR PSPChamps : ARRAY[1..22] OF STRING[35];
    NoChamp   : Byte;
    Touche    : Char;

   PROCEDURE AfficheChampPSP(Indice, Attr : Byte);
   BEGIN
     GotoXy(5, Indice+1); TextAttr := Attr;
     Write(PSPChamps[Indice]);
   END;

   PROCEDURE InitChampsPSP;
   VAR Indice : Byte;
   BEGIN
    PSPChamps[1] := '[00h]  INT 20h
```

Programme MapPSP.Pas (suite) ❹

```
    PSPChamps[2] := '[02h]  Dernier Segment alloué      ';
    PSPChamps[3] := '[04h]  Réservé                 ';
    PSPChamps[4] := '[05h]  Appel FAR aux fonctions DOS  ';
    PSPChamps[5] := '[0Ah]  Ancienne INT 22h        ';
    PSPChamps[6] := '[0Eh]  Ancienne INT 23h        ';
    PSPChamps[7] := '[12h]  Ancienne INT 24h        ';
    PSPChamps[8] := '[16h]  Adresse du PSP père     ';
    PSPChamps[9]  := '[18h]  File handle table      ';
    PSPChamps[10] := '[2Ch]  Bloc d''Environnement  ';
    PSPChamps[11] := '[2Eh]  Sauvegarde de la Pile  ';
    PSPChamps[12] := '[32h]  Nombre de handles (20) ';
    PSPChamps[13] := '[34h]  Adresse de la Fht      ';
    PSPChamps[14] := '[38h]  Prochain PSP (inutilisé)  ';
    PSPChamps[15] := '[3Ch]  Réservé                ';
    PSPChamps[16] := '[50h]  Int 21h + Ret FAR      ';
    PSPChamps[17] := '[53h]  Réservé                ';
    PSPChamps[18] := '[55h]  Extension du FCB n° 1  ';
    PSPChamps[19] := '[5Ch]  FCB n° 1               ';
    PSPChamps[20] := '[6Ch]  FCB n° 2               ';
    PSPChamps[21] := '[7Ch]  Réservé                ';
    PSPChamps[22] := '[80h]  Ligne de Commande      ';
    TextAttr:=15+1*16;
    FOR Indice := 1 TO 22 DO
      AfficheChampPSP(Indice, (15+1*16));
    AfficheChampPSP(1, (14+4*16));
  END;

  FUNCTION Deplace(Lig : Byte) : Byte;
  VAR  Touche : Char;
       No     : Byte;
  BEGIN
    Touche := #215; No := Lig-1;
    WHILE (Touche <> #27) AND (Touche <> #13) DO
    BEGIN
      Touche := ReadKey;
      CASE Touche OF
        #0  : BEGIN
                Touche := ReadKey;
                AfficheChampPSP(No, 15+1*16);
                CASE Touche OF
                  #80 : Inc(No);
                  #72 : Dec(No);
                  #79 : No := 22;
                  #71 : No := 1;
                END; { CASE }
```

➥

Programme MapPSP.Pas (suite) ❺

```
                  IF (No > 22) THEN No := 1
                  ELSE
                  IF (No < 1) THEN No := 22;
                  AfficheChampPSP(No, 14+4*16);
                END;
        #13 : Deplace := No;
      END; { CASE }
    END;
    IF (Touche = #27) THEN
      Deplace := 0;
  END;

  PROCEDURE Ascii(Offset, Taille, Col, Lig : Byte);
  CONST SetOk  : SET OF Char = [#32..#122];
  VAR   Chaine : STRING;
        i      : Byte;
  BEGIN
    Chaine := '';
    FOR i:=0 TO Taille-1 DO
      IF Chr(Mem[AdrPSP:Offset+i]) IN SetOk THEN
        Chaine := Chaine+Chr(Mem[AdrPSP:Offset+i]);
    GotoXy(Col, Lig); TextAttr:=15+7*16;
    Write(' ', Chaine, ' ');
    TextAttr:=1+1*16;
  END;

  PROCEDURE DumpPSP(Offset, Taille, Col, Lig : Byte);
  VAR  Chaine : STRING;
       i      : Byte;
  BEGIN
    Chaine := '';
    IF (Taille = 2) THEN
      Chaine :=  MotDecVersHex(MemW[AdrPSP:Offset])
    ELSE
    IF (Taille = 1) THEN
      Chaine := OctetDecVersHex(Mem[AdrPSP:Offset])
    ELSE
      FOR i:=0 TO Taille-1 DO
        Chaine := Chaine + ' '+
                  OctetDecVersHex(Mem[AdrPSP:Offset+i]);
    GotoXy(Col, Lig); TextAttr := 15+7*16;
    Write(' ',Chaine, ' ');
    TextAttr := 1+1*16;
  END;
```

➥

Programme MapPSP.Pas (suite) ❻

```
BEGIN
  TextAttr:=15+1*16; ClrScr;
  InitChampsPSP; NoChamp := 1;
  WHILE (NoChamp > 0) DO
  BEGIN
    Window(1,1,39,24);
    NoChamp := Deplace(NoChamp+1);
    Window(40,1,80,24);
    Touche := #215;
    CASE NoChamp OF
      0       : BEGIN
                  Window(1,1,80,25);
                  TextAttr:=1+1*16; ClrScr;
                  Exit;
                END;
      1       : BEGIN
                  DumpPSP(0, 2, 1, 1+NoChamp);
                END;
      2       : DumpPSP(2, 2, 1, 1+NoChamp);
      3       : DumpPSP(3, 1, 1, 1+NoChamp);
      4       : DumpPSP(5, 5, 1, 1+NoChamp);
      5       : BEGIN
                  DumpPSP($C, 2, 5, 1+NoChamp);
                  DumpPSP($A, 2, 10, 1+NoChamp);
                END;
      6       : BEGIN
                  DumpPSP($10, 2, 5, 1+NoChamp);
                  DumpPSP($E, 2, 10,  1+NoChamp);
                END;
      7       : BEGIN
                  DumpPSP($14, 2, 5, 1+NoChamp);
                  DumpPSP($12, 2, 10, 1+NoChamp);
                END;
      8       : DumpPSP($16, 2, 5, 1+NoChamp);
      9       : BEGIN
                  Window(40, 1+NoChamp,79, 1+NoChamp+1);
                  TextAttr:=15+7*16; ClrScr;
                  DumpPSP($18, 20, 1, 1+NoChamp);
                END;
      10      : DumpPSP($2C, 2, 5, 1+NoChamp);
      11      : BEGIN
                  DumpPSP($30, 2, 5, 1+NoChamp);
                  DumpPSP($2E, 2, 10, 1+NoChamp);
                END;
```

➡

Programme MapPSP.Pas (suite) ❼

```
     12       : DumpPSP($32, 2, 5, 1+NoChamp);
     13       : BEGIN
                  DumpPSP($36, 2, 5, 1+NoChamp);
                  DumpPSP($34, 2, 10, 1+NoChamp);
                END;
      14      : BEGIN
                  DumpPSP($3A, 2, 5, 1+NoChamp);
                  DumpPSP($38, 2, 10, 1+NoChamp);
                END;
      15      : BEGIN
                  Window(40, 1+NoChamp, 79, 1+NoChamp+2);
                  TextAttr:=15+7*16; ClrScr;
                  DumpPSP($3C, 20, 1, 1+NoChamp);
                END;
      16      : DumpPSP($50, 3, 5, 1+NoChamp);
      17      : DumpPSP($53, 2, 5, 1+NoChamp);
      18      : DumpPSP($55, 7, 5, 1+NoChamp);
      19,20   : BEGIN
                  Window(40, NoChamp-5, 79, NoChamp);
                  TextAttr:=15+7*16; ClrScr;
                  DumpPSP($5C+(NoChamp-19), 16, 1, NoChamp-5);
                  Ascii($5C+(NoChamp-19), 16, 1, NoChamp-3);
                END;
      21      : DumpPSP($7C, 4, 5, 1+NoChamp);
      22      : BEGIN
                  Window(40, NoChamp-10, 79, NoChamp);
                  TextAttr:=15+7*16; ClrScr;
                  DumpPSP($80, 128, 1, 1);
                  Ascii($82, 124, 1, WhereY+1);
                END;
    END; { CASE }
    WHILE (Touche = #215) DO
      Touche := ReadKey;
    TextAttr:=1+1*16; ClrScr;
    Window(1,1,80,25);
  END;
  Window(1,1,80,25);
END;

FUNCTION Deplace(Lig : Byte; Premier : PtrPSP) : Word;
CONST
    Curseur: STRING[6] = '      ';
```

➥

Programme MapPSP.Pas (suite) ❽

```
VAR Actuel, Ex : PtrPSP;
    Touche     : Char;

    PROCEDURE AfficheCurseur(Lig : Byte; Attr : Integer;
    Seg : Word);
    BEGIN
      GotoXy(21, Lig); TextAttr:=Attr; Write(Curseur);
      GotoXy(22, Lig); Write(MotDecVersHex(Seg));
    END;

BEGIN
  Actuel := Premier; Touche := #215;
  WHILE (Touche <> #27) DO
  BEGIN
    AfficheCurseur(Lig, 14+4*16, Actuel^.Segment+1);
    Touche := ReadKey;
    CASE Touche OF
      #27 : Deplace := Actuel^.Segment+1;
      #0  : BEGIN
              AfficheCurseur(Lig, 15+5*16,
                             Actuel^.Segment+1);
              Touche := ReadKey; Ex := Actuel;
              CASE Touche OF
                #80 : BEGIN  { Flèche Bas }
                        IF (Actuel^.Suivant^.Segment <>
                            $FFFF) THEN
                          BEGIN
                            Ex := Actuel;
                            Actuel := Actuel^.Suivant;
                            Inc(Lig);
                          END;
                        END;
                #72 : BEGIN  { Flèche Haut }
                          IF (Actuel <> Premier) THEN
                          BEGIN
                          Ex := Actuel; Actuel := Premier;
                        WHILE Actuel^.Suivant <> Ex DO
                            Actuel := Actuel^.Suivant;
                          Dec(Lig);
                          END;
                        END;
                #79 : BEGIN  { END }
                          Ex := Actuel;
                        WHILE (Actuel^.Suivant^.Segment <>
                               $FFFF) DO
```

➥

Programme MapPSP.Pas (suite) ❾

```
                            Actuel := Actuel^.Suivant;
                           Lig := 5+(NbPSP-1);
                         END;
                 #71 : BEGIN  { Home }
                           Ex := Actuel;
                           Actuel := Premier;
                           Lig := 5;
                         END;
               END; { CASE }
               IF (Lig < 5) THEN
                 Lig := 5
               ELSE
               IF (Lig > NbPSP+5) THEN
                 Lig := NbPSP+5;
               AfficheCurseur(Lig, 14+4*16,
     Actuel^.Segment+1);
             END;
      #13 : BEGIN
               ChampsPSP(Actuel^.Segment+1);
               AfficheAdresses(Liste);
             END;
    END; { CASE }
  END;
  Deplace := Actuel^.Segment+1;
END;

BEGIN
  TextAttr:=15+1*16; ClrScr;
  New(Liste); Liste^.Suivant := NIL; Liste^.Offset := 0;
  IF (PremierPSP <> $FFFF) THEN
    Liste^.Segment := PremierPSP;
  NbPSP := ListePSP(Liste);
  AfficheAdresses(Liste);
  Segment := Deplace(5, Liste);
  Dispose(Liste); Liste := NIL;
  Window(1,1,80,25); TextAttr := 15+1*16; ClrScr;
END.
```

Afficher et modifier la mémoire

Le dernier programme de ce chapitre, `DmpMem.Pas`, est à la fois utile, agréable à utiliser et instructif : il s'agit d'un dumper mémoire. Lorsqu'on l'utilise en combinaison avec les autres utilitaires fournis dans ce chapitre, la mémoire (RAM et ROM) ne peut plus avoir de secrets pour l'utilisateur. Il permet de modifier le contenu de la RAM, ce qui peut servir à la condition d'être prudent : si vous modifiez le driver d'écran, ne vous étonnez pas de devoir relancer votre machine...

Sur les dix-huit procédures et fonctions qu'il utilise (voir la description plus bas), deux seulement en constituent le cœur : ce sont `DumpSecteur` et `LitCar`. La première fait appel à la procédure prédéfinie `Move` et copie 512 octets de la mémoire dans un tableau de chaînes hexadécimales, ce qui permet d'afficher le secteur lu. Comme un écran de Dump comporte aussi une partie réservée à la traduction ASCII des octets renvoyés, `DumpSecteur` intègre une procédure `DumpAsc` qui se charge de cette seconde traduction.

`LitCar` est chargée du déplacement des deux curseurs à l'écran, ce qui ne serait pas une tâche importante si l'on ne pouvait modifier un des octets affichés. C'est en effet la gestion des modifications apportées aux secteurs affichés qui se trouve au cœur du problème. Pour qu'elle se fasse de la façon la plus simple possible, nous avons créé la variable `OctetEnCours`. Il s'agit d'un `RECORD`, qui comporte l'indice du tableau auquel se trouve l'octet pointé par le curseur, ainsi que l'octet lui-même sous forme décimale, hexadécimale et ASCII. Lorsqu'on appuie sur une touche comprise entre "`0`" et "`9`" ou entre "`A`" et "`F`", les valeurs hexadécimale, décimale et ASCII de l'octet sont modifiées, puis le tableau de chaînes et le tableau d'octets sont mis à jour. Dès lors, la seule façon d'éviter que la RAM soit effectivement modifiée est de répondre par la négative au message envoyé lorsqu'on se déplace en mémoire, et que le programme propose de rendre les modifications effectives en recopiant le tableau en RAM. Ce `RECORD` permettra aux amateurs d'étendre les fonctionnalités du programme en autorisant la saisie dans la zone ASCII, ce qui n'a pas été réalisé ici par manque de place. On gère également les déplacements de 512 octets en 512 octets dans cette procédure (plus exactement dans les procédures `PgeUp` et `PgeDn`, qui sont internes à `LitCar`).

Enfin, dernière particularité de la procédure `LitCar` : elle est récursive. Il fallait en effet pouvoir sortir de l'affichage des secteurs d'un disque pour afficher ceux d'un autre disque, ce qui nécessitait d'appeler le Menu, la procédure d'Initialisation, et la procédure de Dump en leur passant de nouveaux paramètres. Pour des raisons qui apparaîtront évidentes à la lecture, c'est `LitCar` qui appelle la procédure chargée de ce processus : la récursivité était presque inévitable. La seule autre possibilité aurait été de déclarer une procédure `FORWARD`, ce qui n'aurait pas été d'une aussi grande efficacité et qui ne serait pas très "pascalien" : deux raisons qui nous l'ont fait éviter.

Procédures internes à DMPMEM :

Type	Nom	Paramètres	Fonction
F	MemoireDispo		Renvoie la taille de la RAM
F	Caps	Chaine (STRING)	Renvoie la chaîne en majuscules
P	Titre		Affiche le titre à l'écran
P	Cadre	Col1, Lig1, Col2, Lig2 (BYTE)	Trace un cadre double aux couleurs en cours
P	Ecran		Trace l'écran principal
P	Menu		Affiche le Menu
P	Barre	Octet (Fntr)	Affiche la barre d'informations
P	DumpSecteur	Drive (BYTE) VAR NoSect (WORD)	Dumpe un secteur (Hex)
P	DumpAsc		Dumpe un secteur (ASCII)
P	AffCurs	Col, Lig, Attr (BYTE), Octet (Fntr)	Affiche le curseur (ou l'efface selon Attr.)
P	Affiche	Lig, Indice (BYTE)	Affiche Seize lignes du tableau de chaînes
P	Erreur	NoErreur (BYTE)	Affiche un message d'erreur
P	PgeUp	VAR Lig	Affiche la page ou le secteur précédent (suivant)
P	PgeDn	Indice (BYTE) Octet (Fntr) VAR Ok (BOOLEAN)	
P	LitCar	Indice (BYTE)	Lit le clavier, modifie le secteur
P	Programme DumpSecteur		Appelle Menu, Init,

Malgré tout, utilisez-le sans compter. Petit à petit, vous apprendrez sur la mémoire bien plus que ne peut vous enseigner ce livre ou n'importe quel autre sur le sujet. Pour savoir programmer, il faut programmer, pour comprendre le système, il faut le regarder, le modifier et faire le plus de recoupements possible entre les diverses sources de renseignements que l'on possède.

Listing 3.22
Programme DmpMem.Pas. ❶

```
PROGRAM DumperMemoire; { DmpMem.Pas }

USES Dos, Crt, Sys;

TYPE
  FauxMot  = 0..($FFFF+16);
  Sect     = ARRAY[1..512] OF Byte;
  DumpSect = ARRAY[1..32] OF STRING[80];
  Fntr     = RECORD
                NoOctet : Word;
                Octet   : Byte;
                Hexa    : STRING[2];
                Ascii   : Char;
             END;

VAR  Secteur             : Sect;
     SectDump            : DumpSect;
     Segment, Offset,
     NOffset             : FauxMot;
     Chaine              : STRING;

FUNCTION MemoireDisponible : Word;
BEGIN
 MemoireDisponible:=MemW[$0040:$0013];
END;

FUNCTION Caps(Chaine : String) : String;
VAR i : Integer;
BEGIN
 FOR i:=1 TO Length(Chaine) DO
   Chaine[i]:=UpCase(Chaine[i]);
 Caps:=Chaine;
END;

PROCEDURE Cadre(Col1, Lig1, Col2, Lig2 : Byte);
VAR i : Integer;
BEGIN
 FOR i:=Lig1 TO Lig2-1 DO
 BEGIN
   GotoXy(Col1, i); Write(#186, '':(Col2-Col1), #186);
 END;
 FOR i:=Col1+1 TO Col2 DO
```

➥

Programme DmpMem.Pas (suite) ❷

```
  BEGIN
    GotoXy(i,Lig1-1); Write(#205);
    GotoXy(i,Lig2); Write(#205);
  END;
  GotoXy(Col1,Lig1-1); Write(#201);
  GotoXy(Col2+1,Lig1-1);
  Write(#187); GotoXy(Col1,Lig2); Write(#200);
  GotoXy(Col2+1,Lig2); Write(#188);
END;

PROCEDURE Titre;
BEGIN
  TextAttr:=14+4*16; Cadre(22, 2, 58, 3);
  GotoXy(24,2);
  Write(' M E M O R Y   D U M P E R   1.0 ');
END;

PROCEDURE Erreur(NoErreur : Byte);
VAR  CarErr : CHAR;
BEGIN
  TextAttr:=15+4*16; Cadre(2,3,77,4); GotoXy(10,3);
  CASE NoErreur OF
    0 :  Write('':15, ' Sauver en mémoire ? (O/N)  : ');
    1 :  Write('':11, ' ERREUR : Adresse Mémoire
               inexistante ! ');
    2 :  Write('':12, ' ERREUR : Impossible d''Écrire en
               ROM ! ');
  END; { Case }
  CarErr:=#215;
  REPEAT
    CarErr:=ReadKey; CarErr:=UpCase(CarErr);
  UNTIL ((CarErr = #27) OR ((CarErr = 'O') AND
        (NoErreur = 0)));
  IF (CarErr = 'O') THEN
    Move(Mem[Seg(Secteur):Ofs(Secteur)],
         Mem[Segment:Offset], SizeOf(Secteur));
  TextAttr:=7+7*16; Cadre(2,3,77,4); Titre;
END;

PROCEDURE Ecran;
BEGIN
  TextAttr:=15+7*16; ClrScr; TextAttr:=14+1*16;
  Cadre(2,7, 77,23); TextAttr:=14+4*16; GotoXy(2,24);
  Write('':77); Titre;
END;
```

➥

Programme DmpMem.Pas (suite) ❸

```
PROCEDURE Menu;
VAR SegMem, OfsMem : STRING;
    Memoire        : LongInt;
    Car            : Char;

BEGIN
 TextAttr:=15+7*16; Cadre(25,12, 55,19); GotoXy(27,13);
 Write('Mémoire RAM : ', MemoireDisponible, ' Ko ');
 GotoXy(WhereX, 13); Write('(',
        MotDecVersHex(MemoireDisponible), 'h)');
 GotoXy(27,15); Write('Segment     : '); GotoXy(27,17);
 Write('Déplacement : '); GotoXy(42,15); ReadLn(SegMem);
 GotoXy(42,17); ReadLn(OfsMem);
 IF ((SegMem = '') OR (OfsMem = '')) THEN
 BEGIN
   TextAttr:=7+1*16; ClrScr; Halt;
 END;
 SegMem:=Caps(SegMem); OfsMem:=Caps(OfsMem);
 Segment:=HexaVersDecimal(SegMem);
 Offset:=HexaVersDecimal(OfsMem);
 Memoire:=LongInt(MemoireDisponible);
 Memoire:=Memoire*1024;
 IF ((LongInt(Segment SHL 4)+Offset) >
     ((Memoire SHL 4) + $FFFF)) THEN
 BEGIN
   Erreur(1); Menu;
 END;
END;

PROCEDURE Barre(Octet : Fntr; NouvOffs : FauxMot);
VAR Chaine : STRING;
   AdrAbs : LongInt;
BEGIN
 TextAttr:=15+4*16; GotoXy(3,24);
 WITH Octet DO
 BEGIN
   Write(' Segment ', MotDecVersHex(Segment),
         ' | Déplacement ');
   Write(MotDecVersHex(NouvOffs),' | Hexa ',Hexa,
         ' | Ascii ',Ascii);
   Write(' | Octet ',Octet:3,' | N° ',NoOctet:3);
 END;
 AdrAbs:=LongInt((Segment SHL 4)+(NouvOffs));
 IF (AdrAbs  <= $3FF) THEN
  Chaine:=' Table des Vecteurs d''Interruptions '
```

➥

Programme DmpMem.Pas (suite) ❹

```
  ELSE IF ((AdrAbs >= $400) AND (AdrAbs <= $4FF)) THEN
   Chaine:=' Données BIOS '
  ELSE IF ((AdrAbs >= $500) AND (AdrAbs <= $6FF)) THEN
   Chaine:=' Données DOS '
  ELSE IF ((AdrAbs >= $700) AND (AdrAbs <= $9FFFF)) THEN
   Chaine:=' RAM '
  ELSE IF ((AdrAbs >= $A0000) AND (AdrAbs <= $AFFFF)) THEN
   Chaine:=' RAM Vidéo Graphique '
  ELSE IF ((AdrAbs >= $B0000) AND (AdrAbs <= $B0FFF)) THEN
   Chaine:=' RAM Vidéo Monochrome Texte '
  ELSE IF ((AdrAbs >= $B1000) AND (AdrAbs <= $B7FFF)) THEN
   Chaine:=' RAM Libre '
  ELSE IF ((AdrAbs >= $B8000) AND (AdrAbs <= $BBFFF)) THEN
   Chaine:=' RAM Vidéo Couleurs Texte '
  ELSE IF ((AdrAbs >= $BC000) AND (AdrAbs <= $BFFFF)) THEN
   Chaine:=' Extension Vidéo '
  ELSE IF ((AdrAbs >= $C0000) AND (AdrAbs <= $C7FFF)) THEN
   Chaine:=' Extension ROM '
  ELSE IF ((AdrAbs >= $C8000) AND (AdrAbs <= $CBFFF)) THEN
   Chaine:=' Disque Dur '
  ELSE IF ((AdrAbs >= $CC000) AND (AdrAbs <= $EFFFF)) THEN
   Chaine:=' Extension ROM '
  ELSE IF ((AdrAbs >= $F0000) AND (AdrAbs <= $F3FFF)) THEN
   Chaine:=' ROM réservée '
  ELSE IF ((AdrAbs >= $F4000) AND (AdrAbs <= $F5FFF)) THEN
   Chaine:=' ROM Inutilisée '
  ELSE IF ((AdrAbs >= $F6000) AND (AdrAbs <= $FDFFF)) THEN
   Chaine:=' ROM Basic '
  ELSE IF ((AdrAbs >= $FE000) AND (AdrAbs <= $FFFFF)) THEN
   Chaine:=' ROM BIOS ';
  GotoXy(1,5); TextAttr:=7+7*16; Write('':79);
  GotoXy((40-(Length(Chaine) DIV 2)), 5);
  TextAttr:=14+4*16; Write(Chaine);
END;

PROCEDURE DumpSecteur;
VAR i, k          : Integer;
    Chaine, Octet: STRING;

 PROCEDURE DumpAsc;
 VAR j : Integer;
 BEGIN                    { DumpAsc }
   Chaine:=Chaine+' | '; j:=(i-16);
```

➥

Programme DmpMem.Pas (suite) ❺

```
   REPEAT
     IF NOT (Secteur[j] In [0..14]) THEN
       Chaine:=Chaine+Chr(Secteur[j])
     ELSE
       Chaine:=Chaine+' ';
     Inc(j);
   UNTIL j > i-1;
   SectDump[k]:=SectDump[k]+Chaine; Chaine:=''; Inc(k);
 END;

BEGIN { Procédure DumpSecteur }
 Chaine:='';
 Move(Mem[Segment:Offset],
      Mem[Seg(Secteur):Ofs(Secteur)], SizeOf(Secteur));
 FOR i:=1 TO 32 DO
   SectDump[i]:=OctetDecVersHex(i-1)+' | ';
 i:=1; k:=1;
 REPEAT
   IF (Length(Chaine) = 48) THEN
     DumpAsc
   ELSE
   IF (Length(Chaine) < 48) THEN
   BEGIN
     Chaine:=Chaine+OctetDecVersHex(Secteur[i])+' ';
     Inc(i);
   END
 UNTIL (i > 512);
 DumpAsc;
END;

PROCEDURE AffCurs(Col,Lig,Attr: Byte; Octet: Fntr);
VAR i : Integer;
BEGIN
 TextAttr:=Attr; GotoXy(Col, Lig); Write(Octet.Hexa);
 GotoXy(((Col-9) DIV 3)+60, Lig); Write(Octet.Ascii);
END;

PROCEDURE Affiche(Lig, Indice : Byte);
VAR i : Integer;
BEGIN
 TextAttr:=15+1*16; i:=Indice;
 REPEAT
   GotoXy(4,Lig); WriteLn(SectDump[i+1]);
   Inc(i); Inc(Lig);
 UNTIL (i > Indice+15);
END;
```

➥

Programme DmpMem.Pas (suite) ❻

```
PROCEDURE LitCar(Indice: Byte);
VAR Car, CarErr          : Char;
    Col, Lig, i, Compteur: Byte;
    OctetEnCours         : Fntr;
    Modif                : BOOLEAN;
    AdrAbs               : LongInt;

 PROCEDURE Programme;
 BEGIN
   Menu; DumpSecteur;
 END;

 PROCEDURE PgeUp(VAR Lig,Indice: Byte;Octet: Fntr;
                VAR Ok: BOOLEAN);
 BEGIN
   IF (Octet.NoOctet > 256) THEN
   BEGIN
     Affiche(7,0); Indice:=0; Lig:=7;
   END
   ELSE
   IF NOT ((Segment SHL 4)+Offset < 0) THEN
   BEGIN
     IF Ok THEN
       Erreur(0);
     IF (Offset >= $200) THEN
     BEGIN
       Dec(Offset,$200); DumpSecteur;
       Affiche(7,0); Indice:=0; Lig:=7; Ok:=FALSE;
     END
     ELSE
       LitCar(0);
   END;
 END;

 PROCEDURE PgeDn(VAR Lig,Indice: Byte;Octet: Fntr;
                VAR Ok: BOOLEAN);
 BEGIN
   IF (Octet.NoOctet <= 256) THEN
   BEGIN
     Affiche(7, 16); Indice:=16; Lig:=7;
   END
   ELSE
   IF (Segment < $FFFF) THEN
```

➥

Programme DmpMem.Pas (suite) ❼

```
    BEGIN
      IF Ok THEN
        Erreur(0);
      IF (NOffset+$200 < $FFFF) THEN
      BEGIN
        Inc(Offset,$200); DumpSecteur;
        Affiche(7,0); Indice:=0; Lig:=7; Ok:=FALSE;
      END
      ELSE
        LitCar(0);
    END;
  END;

BEGIN { Procédure LitCar }
  Programme; Car:=#215; Col:=9; Lig:=7;
  Compteur:=1; Modif:=False;
  Affiche(Lig, Indice);
  WITH OctetEnCours DO
  BEGIN
    NoOctet:=(Indice * 16)+(Col DIV 3)-2;
    Octet:=Secteur[NoOctet];
    Hexa:=SectDump[Indice+1][Col-3] +
    SectDump[Indice+1][Col-2];
    Ascii:=SectDump[Indice+1][((Col-9) DIV 3)+57];
  END;
  NOffset:=Offset+(OctetEnCours.NoOctet-1);
  AffCurs(Col, Lig,(2*16+6), OctetEnCours);
  Barre(OctetEnCours, NOffset);
  REPEAT
    Car:=ReadKey; Car:=UpCase(Car);
    CASE Car OF
    #0 : BEGIN
          Car:=ReadKey;
          AffCurs(Col,Lig,(1*16+15),OctetEnCours);
          CASE Car OF
           #73 : BEGIN
                   PgeUp(Lig,Indice,OctetEnCours,Modif);
                   NOffset:=Offset+((Indice * 16)+
                            (Col DIV 3)-2)-1;
                   Barre(OctetEnCours, NOffset);
                 END;
           #77 : BEGIN
                   Inc(Col,3); Inc(NOffset);
                 END;
```

➥

Programme DmpMem.Pas (suite) ❽

```
          #75 : BEGIN
                  Dec(Col,3); Dec(NOffset);
                END;
          #72 : BEGIN
                  Dec(Lig); Dec(Indice);
                  Dec(NOffset, 16);
                END;
          #80 : BEGIN
                  Inc(Lig); Inc(Indice);
                  Inc(NOffset, 16);
                END;
          #81 : BEGIN
                  PgeDn(Lig,Indice,OctetEnCours,Modif);
                  NOffset:=Offset+((Indice * 16)+
                          (Col DIV 3)-2)-1;
                  Barre(OctetEnCours, NOffset);
                END;
          END;
          IF (Col < 9) THEN
          BEGIN
            Col:=54; Dec(Lig); Dec(Indice);
          END
          ELSE
          IF (Col > 54) THEN
          BEGIN
            Col:=9; Inc(Lig); Inc(Indice);
          END;
          IF (Lig < 7) THEN
          BEGIN
            Lig:=22; Inc(Indice,16);
            NOffset:=Offset+((Indice*16)+
                               (Col DIV 3)-2)-1;
          END
          ELSE
          IF (Lig > 22) THEN
          BEGIN
            Lig:=7; Dec(Indice, 16);
            NOffset:=Offset+((Indice*16)+
                               (Col DIV 3)-2)-1;
          END;
          WITH OctetEnCours DO
          BEGIN
            NoOctet:=(Indice * 16)+(Col DIV 3)-2;
            Octet:=Secteur[NoOctet];
```

➥

Programme DmpMem.Pas (suite) ❾

```
              Hexa:=SectDump[Indice+1][Col-3] +
              SectDump[Indice+1][Col-2];
              Ascii:=SectDump[Indice+1] [((Col-9) DIV 3)+57];
            END;
            IF (NOffset > $FFFF) THEN
              LitCar(0)
            ELSE
            IF (NOffset < 0) THEN
              LitCar(0);
            AffCurs(Col,Lig,(2*16+6),OctetEnCours);
          END;
    #65..#70,
    #48..#57 : BEGIN
                  IF NOT (LongInt((Segment SHL 4)+NOffset) >=
                          $F0000) THEN
                  BEGIN
                  WITH OctetEnCours DO
                  BEGIN
                    Hexa[Compteur]:=Car;
                    Octet:=HexaVersDecimal(Hexa);
                    Ascii:=Chr(Octet);
                    SectDump[Indice+1][Col-3]:=Hexa[1];
                    SectDump[Indice+1][Col-2]:=Hexa[2];
                    SectDump[Indice+1][((Col-9)DIV 3)+57]:=Ascii;
                    Secteur[NoOctet]:=Octet;
                  END;
                  AffCurs(Col,Lig,(2*16+6),OctetEnCours);
                  Modif:=True;
                  IF (Compteur < 2) THEN
                    Inc(Compteur)
                  ELSE
                    Compteur:=1;
                  END
                  ELSE
                    Erreur(2);
                END;
    END;
    Barre(OctetEnCours, NOffset);
  UNTIL (Car=#27);
  LitCar(0);
END;

BEGIN
  Ecran;
  LitCar(0);
END.
```

Conclusion

Nous avons vu toutes les structures DOS de gestion de la mémoire, ou résidentes en mémoire : PSP, MCB, Table des chemins, En-têtes de buffers disques, etc. Cela nous a permis d'entrer à la fois au cœur de la RAM standard et du DOS. De nombreuses fonctions non documentées sont en effet réservées à la gestion de la mémoire. Pour plus de détails, consultez le chapitre 9 (*Fichiers .EXE*) et l'annexe 2 (*Interruptions et fonctions cachées du DOS*).

Chapitre 4

Disques au niveau physique : gestion par le BIOS

La gestion des disques et disquettes est probablement le service le plus complet qu'offre le BIOS après celui de l'affichage graphique. Les 255 (`FFh`) fonctions de l'interruption `13h` sont entièrement consacrées aux disques. En outre, la partie des données BIOS en RAM concernant la mémoire de masse est particulièrement importante. Tout y est enregistré, de la vitesse à laquelle les données sont transmises jusqu'au nombre de drives du PC. On trouve aussi des renseignements intéressants en RAM CMOS, dans la table des paramètres de la disquette en cours d'utilisation ou dans les tables (au maximum deux) de paramètres du (des) disque(s) dur(s).

Si le BIOS est aussi fourni au sujet des disques, c'est bien sûr parce que son rôle d'interface entre le matériel et le système d'exploitation l'y oblige. Mais surtout, c'est à lui que revient la charge de formater physiquement un disque. En outre les fonctions DOS de gestion du disque finissent toujours par un appel BIOS de lecture ou d'écriture...

Disquettes

L'interruption `13h` comporte 11 services documentés de gestion de la disquette. Ces services sont tous très proches du matériel et sont loin d'être aussi élaborés que ceux proposés par l'interruption `21h` du DOS. En revanche, il est quasiment impossible de passer par l'Int `13h` sans utiliser presque toutes ses fonctions : si aucune n'est superflue, aucune n'est inutile non plus.

Fonctions disquette de l'Int 13h

Description	*Numéro*	*Paramètres*	*Sorties*
Réinitialisation du contrôleur et du lecteur	00h	AH := 00h DL := NoDsk	Si CF=1, AH = NoErreur
Lecture de l'état	01h	AH := 01h DL := NoDsk	Si CF=1, AH = NoErreur AL = Etat du disque après la dernière opération
Lecture de NbSect	02h	AH := 02h AL := NbSect CH := NoPste CL := NoSect DH := NoTete DL := NoDsk ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF = 1, AH = NoErreur AL = Nb lus

(suite du tableau)

Description	Numéro	Paramètres	Sorties
Ecriture de NbSect	03h	AH := 03h AL := NbSect CH := NoPste CL := NoSect DH := NoTete DL := NoDsk ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF=1, AH = NoErreur AL = Nb écrits
Vérifier une piste	04h	AH := 04h AL := NbSect CH := NoPste CL := NoSect DH := NoTete DL := NoDsk ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF=1, AH = NoErreur AL = NbSect transférés
Formater une piste	05h	AH := 05h AL := NbSect DH := NoTete DL := NoDsk CH := NoPste ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF=1, AH = NoErreur
Paramètres disquette *AT seulement*	08h	AH := 08h DL := NoDsk	Si CF = 1, AH = NoErreur Si CF = 0, AX = 0000h AX = 0000h BH = 00h BL = TypeDsk CH = MaxPiste CL = MaxSect DH = MaxTete DL = NbDsk ES = Seg(Table de Paramètres) DI = Ofs(Table de Paramètres)
Type du lecteur *AT seulement*	15h	AH := 15h DL := NoDsk	Si CF = 1, Erreur = 0 AH = Type Dsk
Changement de disquette *AT seulement*	16h	AH := 16h DL := NoDsk	Si CF= 1, AH = NoErreur Si CF= 0, AH = résultat
Configurer disquette *AT seulement* (disqueet drive)	17h	AH := 17h AL := Format DL := NoDsk	Si CF = 1, AH = NoErreur

➡

(suite du tableau)

Description	Numéro	Paramètres	Sorties
Configurer disquette pour formatage *AT seulement*	18h	AH := 18h CH := NbPste CL := NbSect DL := NoDsk	Si CF = 1, AH = NoErreur ES =Seg(Table de Param) DI =Ofs(Table de Param)

Tableau 4.1
Les fonctions disquette de l'interruption 13h.

Certaines fonctions de l'Int 13h méritent de plus amples explications que le tableau ne peut fournir. On aura cependant remarqué la cohérence dont elles font preuve : ce sont toujours les mêmes registres qui sont affectés aux mêmes paramètres en entrée et en sortie. AH permet de sélectionner le numéro de fonction désiré et renvoie l'état de la disquette après l'exécution de l'Int. CF indique systématiquement s'il y a eu une erreur ou non, etc.

Remarques et précisions

Il n'y a, à un moment donné, qu'une seule table de paramètres disquette en mémoire. Lorsque le lecteur courant change, la table est automatiquement remise à jour par le BIOS.

Les numéros de lecteurs commencent à 00h (pour A:). Il s'agit là d'un point particulièrement important étant donné que le DOS, lui, numérote souvent (mais pas toujours...) les lecteurs à partir de 01h, le lecteur 00h correspondant au drive en cours.

Si AH renvoie la valeur 80h après un appel aux fonctions 05h, 16h ou 18h, le lecteur de disquette n'existe pas ou ne contient pas de disquette. Si cette valeur est renvoyée par une autre fonction, il s'agit d'une erreur de Time-Out (voir table des codes d'erreurs disquette).

Codes d'erreurs disquette renvoyés par l'Int 13h

Numéro de l'erreur	Description
00h	Pas d'erreur
01h	Numéro de fonction invalide
02h	Adresse introuvable
03h	Disque protégé en écriture
04h	Secteur introuvable
06h	Porte du lecteur ouverte

08h	Erreur DMA
09h	Erreur de limite DMA
0Ch	Type de disquette indisponible
10h	Mauvais CRC
20h	Echec du contrôleur disquette
40h	Erreur de positionnement (Seek)
80h	Dépassement (Time-Out)

Tableau 4.2
Codes d'erreurs.

Fonction 00h En entrée, le bit 7 du registre DL doit être à 0 lorsqu'on réinitialise une disquette et à 1 s'il s'agit d'un disque dur.

Cette fonction réinitialise à la fois la carte contrôleur et le lecteur (drive). On l'utilise lorsqu'un problème d'accès au drive a eu lieu plusieurs fois (généralement Trois) de suite.

Fonction 02h Le buffer pointé par ES:BX doit être suffisamment large pour contenir le nombre de secteurs lus.

Il faut toujours vérifier le contenu du registre AL en sortie : le nombre de secteurs effectivement lus peut être différent du nombre de secteurs dont on a demandé la lecture. Dans ce cas, il faut recommencer l'opération.

Si le BIOS renvoie un code d'erreur, c'est peut-être que le moteur du drive n'était pas en route, ou n'avait pas atteint sa vitesse maximale : il faut donc réinitialiser le lecteur de disquette et recommencer. Ce n'est qu'à la troisième fois que le cycle complet aura eu lieu que l'on pourra être sûr qu'il s'agit bien d'une erreur.

Fonction 03h Les remarques sont les mêmes que pour la fonction 02h, l'une et l'autre fonctionnant en parallèle.

Fonction 04h Il est nécessaire de réinitialiser le lecteur de disquette *avant* d'appeler cette fonction. Si CF est positionné à 1 et AH contient une valeur différente de 0 en sortie, le lecteur ne contient pas de disquette. Pour en être certain, il faut cependant recommencer trois fois de suite.

Le buffer pointé en entrée par ES:BX contient les champs d'adresse des secteurs, et a le même format que celui passé à la fonction 05h (voir schéma).

Fonction 05h Si le lecteur de disquette supporte plusieurs formats (lecteurs 1.2 Mo ou 1.44 Mo), il est nécessaire d'appeler la fonction `17h` (configurer le type de disquette) ou la fonction `18h` (configurer disquette pour le formatage) *avant* d'exécuter l'Int `13h`, fonction `05h`.

Le buffer pointé en entrée par `ES:BX` contient les champs d'adresse des secteurs.

Format du buffer des champs d'adresse

La table des champs d'adresse contient une entrée de quatre octets par numéro de secteur relatif à une piste : il y a donc neuf entrées pour une disquette 360 Ko, 15 pour une disquette 1,2 Mo et 18 pour une disquette 1,44 Mo. Ces entrées sont constituées comme suit :

Numéro de l'octet	*Intitulé*
0	Numéro de la piste
1	Numéro de la tête (à partir de 0)
2	Numéro du secteur
3	Taille du secteur : `00h` = 128 octets `01h` = 256 octets `02h` = 512 octets `03h` = 1024 octets

Exemple Pascal

```
TYPE
  RecAdr                    = RECORD
                                Chmp0, Chmp1,
                                Chmp2, Chmp3 : Byte;
                              END;
  TableauPstes              = ARRAY[1..18] OF RecAdr;

VAR
  Dskte144 : TableauPstes; { Disquette 1.44 Mo }

PROCEDURE InitTab; {Initialiser Tab. avant formatage}
VAR  i  : Integer;
BEGIN
  FOR i := 1 TO 18 DO { 18 sect. de 512 o. / piste }
  BEGIN               { piste 0, Tête 1            }
   Dsktte144[i].Chmp0 := 0; Dsktte144[i].Chmp1 := 1;
   Dsktte144[i].Chmp2 := i; Dsktte144[i].Chmp3 := $02;
  END;
END;
```

Tableau 4.3
Format du buffer des champs d'adresse (Fonction 05h de l'Int 13h)

Fonction 08h Cette fonction renvoie les paramètres du lecteur de disquette spécifié. En sortie, `BL` identifie le type de drive comme suit :

`Bits 4-7`	=	`0`	
`Bits 0-3`	=	`01h`	5 pouces 1/4 ; 360 Ko ; 40 pistes
	=	`02h`	5 pouces 1/4 ; 1,2 Mo ; 80 pistes
	=	`03h`	3 pouces 1/2 ; 720 Ko ; 80 pistes
	=	`04h`	3 pouces 1/2 ; 1,44 Mo ; 80 pistes

La table des paramètres pointée par `ES:DI` indique le format maximum supporté par le drive. Si `BL` était égal à `0`, il faudrait aller vérifier dans cette table avant de considérer que la fonction a échoué.

Fonction 15h C'est la seule fonction disquette de l'Int `13h` qui ne renvoie pas un code d'erreur par l'intermédiaire du registre `AH`. Celui-ci contient une première identification du lecteur.

`AH`	= `00h`	Pas de lecteur installé
	= `01h`	Lecteur ne pouvant pas détecter le changement de format de la disquette (360 Ko ou 720 Ko)
	= `02h`	Lecteur pouvant détecter le changement de format de la disquette (1,2 Mo ou 1,44 Mo)
	= `03h`	Disque dur installé

Fonction 16h Si le lecteur ne peut pas signaler au BIOS un changement d'état (porte fermée puis ouverte), la fonction `16h` renvoie `CF` à `0` et `AH` à `06h`. Si `AH` est à `0`, la porte du lecteur n'a pas été ouverte depuis le dernier accès disque. Si `AH` est à `06h` et que le lecteur peut détecter un changement d'état, la porte a été ouverte.

Avant de faire appel à cette fonction, il vaut mieux exécuter l'Int `13H`, fonction `15h` : si le lecteur de disquette est en mesure de détecter un changement d'état, `AH` contiendra la valeur `02h`.

Fonction 17h La fonction `18h` remplace celle-ci pour les versions du DOS 3.2 et suivantes. Cet appel configure la vitesse de transmission des données au lecteur suivant son format. Le registre `AH` doit contenir en entrée l'une des quatre valeurs suivantes :

`AH`	= `01h`	Disquette 360 Ko dans lecteur 360 Ko
	= `02h`	Disquette 360 Ko dans lecteur 1,2 Mo
	= `03h`	Disquette 1.2 Mo dans lecteur 1,2 Mo
	= `04h`	Disquette 720 Ko dans lecteur 720 Ko

On remarque que les lecteurs 1,44 Mo ne sont pas supportés par cette fonction.

Fonction 18h Il faut appeler cette fonction avant de formater une piste (voir fonction 05h). A l'instar de la précédente, elle sélectionne le taux de transfert des données en fonction du format de la disquette et du type de lecteur. Si AH est différent de 0 au retour, il y a eu un problème (voir codes d'erreurs).

Table de paramètres et données diverses

La table des paramètres du lecteur de disquette en cours (onze octets) est pointée par le vecteur d'interruption 1Eh : on peut donc trouver son adresse exacte en lisant l'adresse qui se trouve en 0000h:0078h sous la forme Déplacement:Segment. Elle est souvent stockée parmi les variables DOS (voir exemple, où on la trouve en 0000h:0522h). Le BIOS la remet à jour dès qu'un changement de lecteur se produit. Son format est donné à la suite de l'exemple.

Trouver l'adresse de la table des paramètres disquette

1. Avec DEBUG

```
C:\>debug
-d 0000:0078 l 4
0000:0070                        22 05 00 00        "...
-d 0000:0522 l 0B
0000:0520     DF 02 25 02 12 1B-FF 54 F6 01 08 ..%....T...
-q
```

2. En Turbo Pascal

```
BEGIN
  OfsParamDte := MemW[$0000:$0078];
  SegParamDte := MemW[$0000:$007A];
  FOR i := 0 TO $A DO
   Write(Mem[SegParamDte:OfsParamDte + i],' ');
END.

 223 2 37 2 18 27 255 84 246 1 8
```

Format de la table des paramètres

Déplacement	*Description*
00h	Bits 7-4 : type du drive par rapport au taux de transfert Bits 3-0 : temps de déchargement de la tête
01h	Bits 7-1 : temps de chargement de la tête Bit 0 : mode non-DMA (toujours à 1)

02h	Temps d'arrêt du moteur en tics horloge
03h	Octets par secteur (00=128, 01=256, etc.)
04h	Secteurs par piste (08h, 09h, 0Fh, 12h)
05h	Gap inter-secteurs
06h	Taille des données : sans signification, toujours à FFh
07h	Gap pour formatage
08h	Octet écrit par le formatage : 00h ou F6h
09h	Temps de positionnement des têtes en millisecondes
0Ah	Temps dont a besoin le moteur pour tourner à sa vitesse (en 8° de seconde) : 08h.

Tableau 4.4
Format de la table des paramètres

Les données qui se trouvent en RAM CMOS concernent le type des drives 0 et 1 (A: et B:) et ont le format suivant :

Données drives en RAM CMOS (AT)

Adresse	*Description*	
10h	Bits 7-4 :	Type du drive 0
	0000 =	Pas de drive
	0001 =	Drive 360 Ko
	0010 =	Drive 1.2 Mo
	0011 =	Drive 720 Ko
	0100 =	Drive 1,44 Mo
	Bits 3-0 :	Type du drive 0
	0000 =	Pas de drive
	0001 =	Drive 360 Ko
	0010 =	Drive 1.2 Mo
	0011 =	Drive 720 Ko
	0100 =	Drive 1,44 Mo

Tableau 4.5
Données RAM CMOS sur les lecteurs de disquette.

On accède à la RAM CMOS par l'intermédiaire du port d'adresse 0070h, les données se trouvant au port 0071h. Le listing Pascal qui suit montre comment obtenir les renseignements sur les disques A: et B: en lisant sur les ports d'entrées/sorties concernés.

Listing 4.6
Programme CMOS RAM.

```
PROGRAM CMOSRAM;

USES Sys;

VAR Cmos:  Byte;

PROCEDURE Selectionne(Quartet : Byte);
BEGIN
  CASE  Quartet OF
    0 : WriteLn(' Aucun ');
    1 : WriteLn(' 360  Ko');
    2 : WriteLn(' 1.2  Mo');
    3 : WriteLn(' 720  Ko');
    4 : WriteLn(' 1.44 Mo');
  END;
END;

BEGIN
  IF AT THEN { Seuls les AT ont une RAM CMOS }
  BEGIN
    CLI;     { Interdire les interruptions }
    Port[$0070]:=$10;
    Cmos := Port[$0071];
    STI;     { Ré-autoriser les interruptions }
    WriteLn('Drive A: ');
      { Masquage : garder le quartet fort et décaler }
    Selectionne(Cmos AND $F0 SHR 4);
    WriteLn('Drive B: ');
      { Masquage : garder le quartet faible }
    Selectionne(Cmos AND $0F);
  END;
END.
```

Les données BIOS en RAM concernant les disquettes sont particulièrement nombreuses. Notre programme d'exemple en fournit la signification et nous en avons donné le format au chapitre 2 (*Données du BIOS en RAM*). Nous passons donc directement au programme d'exemple. Il affiche à l'écran les données BIOS ainsi que la table des paramètres disquette.

Listing 4.7
Programme ParmDskt.Pas. ❶

```
PROGRAM ParametresDsktte;     {ParmDskt.Pas}

{ $R+ }

USES Dos, Crt, Sys;

TYPE
  Table = ARRAY[$0..$0A] OF Byte;
  Chmps = RECORD
           TChgtTete,
           TDchgtTete,
           ModeNonDMA,
           TArretMoteur,
           TPositTetes,
           TVitesseMoteur,
           OctetsPSect,
           NbSectPPiste,
           GapPSect,
           GapPSectFormat,
           ValeurFormat    : STRING;
         END;

VAR  TableParam  : Table;
     TableBios   : ARRAY[1..30] OF STRING[60];
     ChampsParam : Chmps;

FUNCTION SegTableParam : Word;
VAR Regs : Registers;
BEGIN
  SegTableParam := MemW[$0:($1E * 4)+2];
END;

FUNCTION OfsTableParam : Word;
VAR Regs : Registers;
BEGIN
  OfsTableParam := MemW[$0:($1E * 4)];
END;

PROCEDURE InitChampsBios;
VAR Tempo : STRING;
    Octet : Byte;
    i     : Integer;
 FUNCTION Message(NoErreur : Byte) : STRING;
```

➥

Programme ParmDskt.Pas (suite). ❷

```
  BEGIN
    CASE NoErreur OF
      $00 : Message := '(Pas d''erreur)';
      $80 : Message := 'Lecteur non prêt';
      $40 : Message := 'Erreur de positionnement';
      $20 : Message := 'Erreur du contrôleur disquette';
      $10 : Message := 'Erreur de CRC à la lecture';
      $0C : Message := 'TYPE de disquette inconnu';
      $09 : Message := 'Erreur de dépassement DMA';
      $08 : Message := 'Opération DMA impossible';
      $06 : Message := 'Porte du lecteur ouverte';
      $04 : Message := 'Secteur introuvable';
      $03 : Message := 'Erreur de protection en écriture';
      $02 : Message := 'Adresse introuvable';
      $01 : Message := 'Appel de fonction illégal';
    END; { CASE }
  END;

  FUNCTION Decode(Octet : Byte) : STRING;
  BEGIN
    CASE Octet OF  { fonctions de conversion dans SYS.TPU }
      00 : Decode := MotDecVersHex(500) + 'h Kb/s';
      01 : Decode := MotDecVersHex(300) + 'h Kb/s';
      02 : Decode := '00' + OctetDecVersHex(250) + 'h Kb/s'
    END; { CASE }
  END;

BEGIN
  FOR i:=1 TO 30 DO
    TableBios[i] := '';
  Octet := (Mem[$0040:$010] AND $C0) SHR 6 + 1;
  Str(Octet, Tempo);
  TableBios[1] := ' '+Tempo+ ' Lecteur(s) de Disquettes ';
  TableBios[2] := ' Disquette bootable          : ' +
     Boole(Mem[$0040:$0010] AND 1);
  TableBios[3] := ' Lecteur A: recalibré        : ' +
     Boole(Mem[$0040:$003E] AND 1);
  TableBios[4] := ' Lecteur B: recalibré        : ' +
     Boole(Mem[$0040:$003E] AND 2);
  TableBios[5] := ' Opération courante          : ';
  IF ((Mem[$0040:$003F] AND $80) SHR 7) = 0 THEN
    TableBios[5] := TableBios[5]+'Lecture ou Vérification'
  ELSE
```

➥

Programme ParmDskt.Pas (suite). ❸

```
IF ((Mem[$0040:$003F] AND $80) SHR 7) = 1 THEN
   TableBios[5] := TableBios[5] + 'Ecriture ou Formatage';
Octet := (Mem[$0040:$003F] AND $30); Tempo:=Chr(Octet+65)
TableBios[6]:=' Le Lecteur '+Tempo+ ': est sélectionné';
TableBios[7] := ' Moteur de A: est ON          : ' +
   Boole(Mem[$0040:$003F] AND 1);
TableBios[8] := ' Moteur de B: est ON          : ' +
   Boole(Mem[$0040:$003F] AND 2);
Tempo := OctetDecVersHex(Mem[$0040:$0040]) + 'h';
TableBios[9]:=' Valeur de Time out du moteur : '+Tempo;
Tempo := Message(Mem[$0040:$0041]);
TableBios[10]:=' Erreur                      : '+Tempo;
Tempo := OctetDecVersHex(Mem[$40:$42]) + ' '+
         OctetDecVersHex(Mem[$40:$43])+' ' +
         OctetDecVersHex(Mem[$40:$44]) + ' ' +
         OctetDecVersHex(Mem[$40:$45]) + ' ' +
         OctetDecVersHex(Mem[$40:$46]) + ' ' +
         OctetDecVersHex(Mem[$40:$47]) + ' ' +
         OctetDecVersHex(Mem[$40:$48]) + ' ' +
         OctetDecVersHex(Mem[$40:$49]) + 'h';
TableBios[11]:=' Octets d''état du contrôleur : '+Tempo;
IF AT THEN     { Fonction Booleenne AT dans SYS.TPU }
BEGIN
  Octet := (Mem[$0040:$008B] AND $C0) SHR 6;
  Tempo := Decode(Octet);
  TableBios[12]:=' Dernière vitesse de transfert de A : '
               + Tempo;
  Octet := (Mem[$0040:$008B] AND $0C) SHR 3;
  Tempo := Decode(Octet);
  TableBios[13]:=' Vitesse de transfert de A au début : '
               + Tempo;
  Octet := (Mem[$0040:$008C] AND $C0) SHR 6;
  Tempo := Decode(Octet);
  TableBios[14]:=' Dernière vitesse de transfert de B : '
               + Tempo;
  Octet := (Mem[$0040:$008C] AND $0C) SHR 3;
  Tempo := Decode(Octet);
  TableBios[15]:=' Vitesse de transfert de B au début : '
               + Tempo;
  Octet := (Mem[$0040:$008F] AND $40) SHR 6;
  TableBios[16]:=' Lecteur B utilisé                  : '
               + Boole(Octet);
  Octet := (Mem[$0040:$008F] AND $20) SHR 5;
  TableBios[17]:=' Lecteur B multirate                : '
               + Boole(Octet);
```

➥

Programme ParmDskt.Pas (suite). ❹

```
    Octet := (Mem[$0040:$008F] AND $10) SHR 4;
    TableBios[18]:=' Lecteur B supporte divers formats  : '
                    + Boole(Octet);
    Octet := (Mem[$0040:$008F] AND 4) SHR 2;
    TableBios[19]:=' Lecteur A utilisé                  : '
                    + Boole(Octet);
    Octet := (Mem[$0040:$008F] AND 2) SHR 1;
    TableBios[20]:=' Lecteur A multirate                : '
                    + Boole(Octet);
    Octet := (Mem[$0040:$008F] AND 1);
    TableBios[21]:=' Lecteur A supporte divers formats  : '
                    + Boole(Octet);
    Octet := (Mem[$0040:$0090] AND $C0) SHR 6;
    Tempo := Decode(Octet);
    TableBios[22]:=' Vitesse de transfert pour A        : '
                    + Tempo;
    Octet := (Mem[$0040:$0090] AND $10) SHR 4;
    TableBios[23]:=' Disquette reconnaissable en A      : '
                    + Boole(Octet);
    Octet := (Mem[$0040:$0091] AND $C0) SHR 6;
    Tempo := Decode(Octet);
    TableBios[24]:=' Vitesse de transfert pour B        : '
                    + Tempo;
    Octet := (Mem[$0040:$0091] AND $10) SHR 4;
    TableBios[25]:=' Disquette reconnaissable en B      : '
                    + Boole(Octet);
    Tempo := OctetDecVersHex(Mem[$0040:$0094]) + 'h';
    TableBios[26]:=' N° de la piste courante pour A     : '
                    + Tempo;
    Tempo := OctetDecVersHex(Mem[$0040:$0095]) + 'h';
    TableBios[27]:=' N° de la piste courante pour B     : '
                    + Tempo;
 END;
END;

PROCEDURE InitChampsParam;
VAR Tempo : Word;
BEGIN
 Move(Mem[SegTableParam:OfsTableParam],
     Mem[Seg(TableParam):Ofs(TableParam)], $0B);
 WITH ChampsParam DO
 BEGIN
   ValeurFormat := OctetDecVersHex(TableParam[8])+'h';
   GapPSectFormat := OctetDecVersHex(TableParam[7])+'h';
```

➥

Programme ParmDskt.Pas (suite). ❺

```
    GapPSect := OctetDecVersHex(TableParam[5])+'h';
    Str(TableParam[4], NbSectPPiste);
    NbSectPPiste := NbSectPPiste+'d';
    Tempo := PuissanceDeux(TableParam[3]);
    Tempo := Tempo * $80;
    Str(Tempo, OctetsPSect);
    OctetsPSect := OctetsPSect + 'd';
    ModeNonDMA := OctetDecVersHex(TableParam[1] AND 0);
    Str(TableParam[9], TPositTetes);
    TPositTetes := TPositTetes + '  ms';
    Str((TableParam[$0A] SHR 3), TVitesseMoteur);
    TVitesseMoteur := TVitesseMoteur + '  ms';
    Str(((TableParam[0] AND $0F) SHL 4), TDchgtTete);
    TDchgtTete:=TDchgtTete + '  ms';
    Str(((TableParam[1] OR $0) SHL 2), TChgtTete);
    TChgtTete := TChgtTete + '  ms';
    TArretMoteur := OctetDecVersHex(TableParam[2])+'h';
  END;
 END;

 PROCEDURE AfficheChamps;
 VAR i, j : Integer;
 BEGIN
  GotoXy(23,5); TextAttr := 15+4*16;
  Write('Table de Paramètres en : ',
         MotDecVersHex(SegTableParam));
  Write(':', MotDecVersHex(OfsTableParam));
  Window(10,6,70,18); TextAttr:=14+4*16; ClrScr;
  WITH ChampsParam DO
  BEGIN
   WriteLn;
   Write('  Chargement des têtes                       :');
   WriteLn('    ', TChgtTete);
   Write('  Déchargement des têtes                     :');
   WriteLn('    ',TDchgtTete);
   Write('  Mode DMA (0 = oui)                         :');
   WriteLn('   ', ModeNonDMA);
   Write('  Positionnement des têtes                   :');
   WriteLn('    ', TPositTetes);
   Write('  Tics horloge avant l''arrêt du moteur      :');
   WriteLn('   ',TArretMoteur);
   Write('  Attente avant entrée/sortie                :');
   WriteLn('    ',TVitesseMoteur);
   Write('  Nbre d''octets par secteur                 :');
   WriteLn('  ', OctetsPSect);
```

➥

Programme ParmDskt.Pas (suite). ❻

```
   Write('  Nbre de secteurs par piste                :');
   WriteLn('   ', NbSectPPiste);
   Write('  Octet écrit par le programme de formatage :');
   WriteLn('    ',ValeurFormat);
   Write('  Gap entre secteurs au formatage           :');
   WriteLn('   ',GapPSectFormat);
   Write('  Gap entre secteurs                        :');
   WriteLn('   ',GapPSect);
  END;
  ReadLn;
  Window(1,1,80,25); GotoXy(1,5); TextAttr:=1+1*16;
  Write('':79); GotoXy(28,5); TextAttr:=15+4*16;
  Write(' Données du BIOS en RAM '); Window(10,6,70,18);
  TextAttr := 14+4*16; ClrScr; WriteLn;
  {$R-}
  FOR i := 1 TO 27 DO
  BEGIN
    WriteLn('':2,TableBios[i]);
    IF (i MOD 10 = 0) THEN
    BEGIN
      ReadLn;
      ClrScr;
      WriteLn;
    END;
  END;
  {$R+}
  ReadLn;
 END;

 BEGIN
   TextAttr:=15+1*16; ClrScr;
   InitChampsParam;
   InitChampsBios;
   AfficheChamps;
 END.
```

Ce programme a ceci d'intéressant qu'il dévoile la plupart des valeurs concernant les lecteurs de disquettes conservées par le BIOS. On peut lui reprocher de ne pas faire appel aux interruptions : cela aurait fourni des résultats plus exploitables, mais aurait également été plus long. En outre, les interruptions sont largement utilisées dans le programme de formatage physique que nous présentons à la fin du chapitre. Ce programme a été conçu pour être utilisé, mais surtout pour être modifié : ne vous en privez pas.

Disques fixes

23 fonctions de l'Int 13h concernent les disques fixes. Certaines d'entre elles sont similaires à celles qui sont orientées disquette : nous ne les détaillerons donc pas. D'autres, plus nombreuses, sont spécifiques aux disques fixes et méritent qu'on s'y attarde. Il faut cependant savoir que ces fonctions spécifiques sont en fait pointées par le vecteur d'interruption 40h et non pas 13h. Lorsqu'un programme appelle une fonction disque fixe de l'Int 13h, celle-ci le redirige sur l'Int 40h. Cette particularité est intéressante lorsque l'on désassemble les interruptions à l'aide de DEBUG (ou de n'importe quel autre désassembleur). En revanche, on ne doit pas en tenir compte pour l'écriture de programmes : c'est toujours l'Int 13h qui doit être appelée.

Fonctions disque fixe de l'Int 13h

Description	*Numéro*	*Paramètres*	*Sorties*
Réinitialisation	00h	AH := 00h	Si CF = 1, AH = NoErreur
Disquette et disque fixe Lecture de l'état	01h		
Lecture de NbSect	02h	AH := 02h AL := NbSect CH := NoCyl CL (7-6) := NoCyl CL (5-0) := NoSect DH := NoTete DL := NoDsk ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF = 1, AH = NoErreur AL = NbLus
Ecriture de NbSect	03h	AH := 03h AL := NbSect CH := NoCyl CL (7-6) := NoCyl CL (5-0) := NoSect DH := NoTete DL := NoDsk ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF = 1, AH = NoErreur AL = NbEcrits

(suite du tableau)

Description	***Numéro***	***Paramètres***	***Sorties***
Vérification NbSect	04h	AH := 04h AL := NbSect CH := NoCyl CL (7-6) := NoCyl CL (5-0) := NoSect DH := NoTete DL := NoDsk	Si CF = 1, AH = NoErreur AL = NbVérifiés
Formater cylindre	05h	AH := 05h AL := Entrelacement CH := NoCyl CL (7-6) := NoCyl CL (5-0) := NoSect DH := NoTete DL := NoDsk ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF = 1, AH = NoErreur
Formater piste *défectueuse XT seulement*	06h	AH := 06h AL := Entrelacement CH := NoCyl DH := NoTete DL := NoDsk	Si CF = 1, AH = NoErreur
Formater lecteur *XT seulement*	07h	AH := 07h AL := Entrelacement CH := NoCyl DL := NoDsk	Si CF = 1, AH = NoErreur
Paramètres disque fixe	08h	AH := 08h DL := NoDsk	Si CF = 1, AH = NoErreur AL = 00h CH = NbCyl CL (7-6) = NbCyl CL (5-0) = NbSect DH = NbTete DL = NbDsk ES = Seg(Table de Param.Dsk) DI = Ofs(Table de Param.Dsk)
Initialise paramètres	09h	AH := 09h DL := NoDsk	Si CF = 1, AH = NoErreur

➡

(suite du tableau)

Description	Numéro	Paramètres	Sorties
Lire secteurs longs	0Ah	AH := 0Ah AL := NbSect CH := NoCyl CL (7-6) := NoCyl CL (5-0) := NoSect DH := NoTete DL := NoDsk ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF = 1, AH = NoErreur
Ecrire secteurs longs	0Bh	AH := 0Bh AL := NbSect CH := NoCyl CL (7-6) := NoCyl CL (5-0) := NoSect DH := NoTete DL := NoDsk ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF = 1, AH = NoErreur
Aller à cylindre	0Ch	AH := 0Ch CH := NoCyl CL (7-6) := NoCyl CL (5-0) := NoSect DH := NoTete DL := NoDsk	Si CF = 1, AH = NoErreur
Réinitialisation du disque fixe	0Dh	AH := 0Dh DL := NoDsk	Si CF = 1, AH = NoErreur
Lire le buffer de test *XT seulement*	0Eh	AH := 0Eh DL := NoDsk ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF = 1, AH = NoErreur
Ecrire le buffer de test *XT seulement*	0Fh	AH := 0Fh DL := NoDsk ES := Seg(Buffer) BX := Ofs(Buffer)	Si CF = 1, AH = NoErreur
Teste si le disque est prêt	10h	AH := 10h DL := NoDsk	Si CF = 1, AH = NoErreur
Recalibrer le disque fixe	11h	AH := 11h DL := NoDsk	Si CF = 1, AH = NoErreur

➥

(suite du tableau)

Description	Numéro	Paramètres	Sorties
Tester la RAM du contrôleur de disque *XT seulement*	12h	AH := 12h AL := NbSect CH := NoCyl CL := NoSect DH := NoTete DL := NoDsk	Si CF = 1, AH = NoErreur AL = 00h
Tester le contrôleur *XT seulement*	13h	AH := 13h AL := NbSect CH := NoCyl CL := NoSect DH := NoTete DL := NoDsk	Si CF = 1, AH = NoErreur AL = 00h
Tester le contrôleur	14h	AH := 14h AL := NbSect CH := NoCyl CL := NoSect DH := NoTete DL := NoDsk	Si CF = 1, AH = NoErreur AL = 00h
Lire le type du disque fixe *AT seulement*	15h	AH := 15h DL := NoDsk	Si CF = 1, AH = NoErreur CX = NbSect

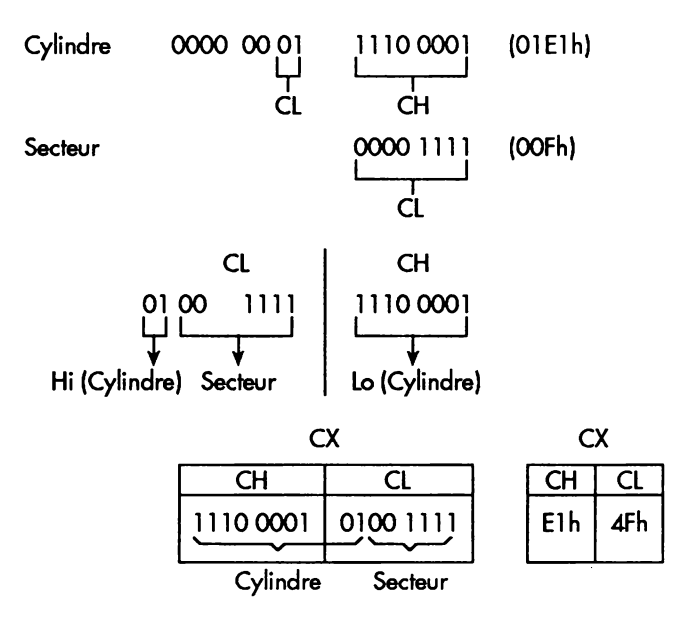
Tableau 4.8
Fonctions disque fixe de l'Int 13h.

Remarques et précisions

Comme dans le cas des fonctions disquette, la cohérence des affectations de registres aux paramètres est à remarquer.

Les numéros de disque dur correspondent au *matériel* et non pas aux disques logiques reconnus par le DOS : on peut tout à fait n'avoir qu'un seul disque dur et trois partitions logiques (notées `C:`, `D:` et `E:`). Ces numéros commencent à `80h`. Les numéros de cylindre et les numéros de tête commencent à `0`.

Dans de nombreuses fonctions, le numéro de cylindre est à exprimer sur 10 bits : `CH` contient les 8 de poids faible, tandis que les bits `7` et `6` de `CL` contiennent les 2 de poids fort. Si l'on veut accéder au secteur `15` (`0Fh`) du cylindre `481` (`01E1h`), on aura donc :



Ce qui donnerait en Pascal :

```
..
VAR Regs : Registers;
WITH Regs DO
BEGIN                  { 8 bits de poids faible }
  CH := Lo($01E1); { 2 bits de poids fort    }
  CL := (Hi($01E1) SHL 6)+$0F; {  plus n°    }
END;                        { de secteur }
```

Codes d'erreur des fonctions disques de l'Int 13h

N° de l'erreur	*Description*
00h	Pas d'erreur
01h	Paramètre ou numéro de fonction invalide
02h	Champ d'adresse introuvable
04h	Secteur introuvable
05h	Echec de la réinitialisation
07h	Numéro de lecteur invalide
08h	Erreur DMA
09h	Erreur de limite

➡

(suite du tableau)

N° de l'erreur	**Description**
0Ah	Mauvais drapeau de secteur
0Bh	Mauvais cylindre
0Dh	Nombre de secteurs invalide
0Eh	Champ d'adresse des données de contrôle trouvé
0Fh	Niveau DMA hors-limites
10h	Erreur ECC ou CRC incorrigible
11h	Erreur : donnée ECC corrigée
20h	Echec du contrôleur
40h	Erreur de positionnement
80h	Dépassement (Time-out)
AAh	Lecteur non prêt
BBh	Erreur indéfinie
CCh	Erreur d'écriture sur le lecteur
E0h	Statut : registre d'erreur à 0
FFh	Echec de l'opération "Sense"

Tableau 4.9
Codes d'erreurs.

Fonction 00h Cette fonction réinitialise à la fois les lecteurs de disquettes et ceux de disque dur. Pour ne réinitialiser que le disque dur, il faut employer la fonction 0Dh.

Fonction 01h A l'exception du registre DL, qui doit contenir la valeur 80h ou 81h, cette fonction est similaire à la fonction disquette 01h (voir le *tableau 4.1 : Les fonctions disquette de l'interruption 13h).*

Fonction 02h Le nombre de secteurs à lire doit être compris entre 1 et 128. Le buffer pointé par ES:BX en entrée doit être suffisamment grand pour contenir les secteurs lus.

Il faut s'assurer que le nombre de secteurs effectivement lus correspond au nombre de secteurs dont on a demandé la lecture. Si la fonction échoue, il faut réinitialiser le lecteur de disque dur et recommencer jusqu'à trois fois de suite.

Fonction 03h Voir les remarques concernant la fonction 02h.

Fonction 04h Cette fonction vérifie que les secteurs spécifiés peuvent être lus et que le CRC est correct. Le programmeur devra s'assurer que le nombre de secteurs effectivement vérifiés correpond bien au nombre de secteurs dont il a demandé la vérification.

Fonction 05h Le facteur d'entrelacement ne doit être spécifié que pour les PC-XT, pas pour les AT, les 386 et les 486. En revanche, la table d'adresses doit tenir compte de ce facteur d'entrelacement.

Format de la table d'adresse pour le formatage disque

La table des champs d'adresse contient une entrée de deux octets par secteur. Il y a donc 17 entrées de deux octets pour un cylindre classique (17 secteurs / piste : un cylindre étant égal à la somme des pistes de même numéro (voir le chapitre 5 pour plus d'informations sur les cylindres).

Numéro de l'octet	Intitulé
0	Drapeau d'état du secteur : 00h : le secteur est bon 80h : le secteur est mauvais
1	Numéro du secteur

Les numéros de secteurs doivent prendre en compte le facteur d'entrelacement.

Exemple Pascal

```
PROGRAM InitPourFormatage;       [InitFrmt.Pas]

USES Sys; { Voir Annexe }
TYPE
  RecFormat  =     RECORD
                           Flag,
                           NoSect : Byte;
                   END;
  TabPstes   =     ARRAY[1..17] OF RecFormat;
VAR
 TabAdresses : TabPstes;
 i           : Integer;
PROCEDURE InitTab;
VAR i, No : Integer;
BEGIN
  FillChar(TabAdresses, SizeOf(TabAdresses), 0);
  IF AT THEN { Fonction Booléenne AT dans SYS.TPU }
  BEGIN
    FOR i := 1 TO 17 DO { Entrelacement 1:1 }
    BEGIN
      TabAdresses[i].Flag   := $00;
      TabAdresses[i].NoSect := i;
    END;

  END
  ELSE                   { PC-XT à Entrelct. 1:2 }
```

```
  BEGIN
    i := 1; No := 1;
    WHILE (No <= 17) DO
    BEGIN
      IF (i MOD 2 = 0) THEN    { Premier Passage }
      BEGIN
        TabAdresses[i].NoSect := No;
        Inc(i,2); Inc(No); { i:=i+2 => on revient là }
      END
      ELSE
      BEGIN { Deuxième Passage }
        TabAdresses[i].NoSect := No;
        Inc(i,2); Inc(No); { i:=i+2 => on revient là }
        IF i > 17 THEN
          i := 2;
      END;
      TabAdresses[i].Flag := $00;
    END;
  END;
END;
       { Programme principal }
BEGIN
  InitTab;
  {$R+} { Impératif sous peine de plantage }
  FOR i := 1 TO 17 DO
    WriteLn(' Indice : ',i:2,' Secteur : ',
      TabAdresses[i].NoSect:2, ' Flag : ',
      TabAdresse[i].Flag);
END.
```

Tableau 4-10
Format de la table d'adresse pour le formatage d'un disque (fonction 05h de l'int 134).

Fonction 06h Cette fonction est spécifique aux XT. Si le code de retour (AH) est 07h, le numéro de lecteur est invalide.

Fonction 07h Cette fonction est spécifique aux XT. Les disques durs de PC-AT doivent être formatés à l'aide de la fonction 05h. Si le code de retour renvoyé par le registre AH est égal à 07h, le numéro du lecteur est invalide.

Fonction 08h En toute logique, cette fonction devrait être appelée avant n'importe quelle autre (sauf celle de réinitialisation) dans un programme concernant le(s) disque(s) dur(s). Elle fournit en effet tous les paramètres réclamés par les autres fonctions de l'Int 13h.

Fonction 09h Cette fonction ne peut renvoyer que deux codes par l'intermédiaire du registre AH: 00h (pas d'erreur) ou 07h (erreur).

Fonction 0Ah Contrairement à la fonction 02h, celle-ci ne corrige pas le Code de Correction d'Erreur (ECC). On l'utilise donc pour les programmes de diagnostic du disque et non pas pour lire un secteur.

Fonction 0Bh Voir les remarques concernant la fonction 0Ah.

Fonction 0Ch Cette fonction positionne les bras du lecteur de disque dur sur le cylindre dont le numéro a été spécifié en CH:CL. Il est inutile de l'appeler avant d'exécuter un appel aux fonctions 02h, 03h, 0Ah, ou 0Bh : celles-ci y font automatiquement appel.

Fonction 0Dh Cette fonction réinitialise le lecteur et le contrôleur du disque fixe spécifié en DL. Contrairement à la fonction 00h, elle ne réinitialise pas les lecteurs et les contrôleurs associés aux disquettes. A la suite d'un appel à cette fonction, les bras du lecteur de disque dur sont positionnés sur le cylindre 0.

Fonction 0Eh Cette fonction est spécifique aux XT. Elle lit le buffer du contrôleur de disque dur et le transfère dans le buffer spécifié par ES:BX.

Fonction 0Fh Cette fonction est spécifique aux XT. C'est le pendant de la précédente. Elle doit être appelée avant de formater un disque dur XT à l'aide de la fonction 05h.

Fonction 10h Cette fonction détermine si le disque dur est prêt à lire ou écrire.

Fonction 11h Cette fonction positionne les bras du disque dur sur le cylindre 0.

Fonction 12h Cette fonction est spécifique aux XT. Elle demande au contrôleur du disque de vérifier ses buffers internes.

Fonction 13h Cette fonction est spécifique aux XT. Elle demande au contrôleur d'effectuer une série de tests sur le disque dur.

Fonction 14h Cette fonction est similaire aux fonctions 12h et 13h, mais est compatible avec les disques AT et les disques XT.

Fonction 15h Cette fonction est spécifique aux AT. Elle renvoie le nombre de secteurs de 512 octets contenus par le disque.

Table de paramètres et données diverses

L'adresse de la table des paramètres du disque dur n° 1 se trouve au vecteur d'interruption 41h, et au vecteur 46h pour le disque dur n° 2 (cas des AT seulement). La table des paramètres disque se trouve en ROM-BIOS. Son format diffère selon qu'il s'agit d'un PC-XT ou d'un AT.

Format des tables de paramètres disque

1. Cas des PC-XT

Déplacement	*Description*	
00h	Nombre de cylindres	
02h	Nombre de têtes	
03h	Cylindre courant	
05h	Cylindre de précompensation	
07h	ECC	
08h	Octet de contrôle :	
	Bit 7 = 1	Pas de rééssais
	Bit 6 = 1	Pas de rééssais ECC
	Bits 5-3 = 0	(Réservés)
	Bits 2-0 =	Option lecteur
09h	Valeur de dépassement (Time-out)	
0Ah	Dépassement pour formatage	
0Bh	Dépassement pour vérification lecteur	
0Ch	Réservé (4 octets)	

2. Cas des PC-AT

Déplacement	*Description*	
00h	Nombre de cylindres	
02h	Nombre de têtes	
03h	Réservé	
05h	Cylindre de précompensation	
07h	Réservé	
08h	Octet de contrôle :	
	Bit 7 = 1	Pas de rééssais
	Bit 6 = 1	Pas de rééssais ECC
	Bit 5 = 1	Carte des défauts du disque sur le dernier cylindre
	Bit 4 = 0	(Réservé)
	Bit 3 = 1	Plus de huit têtes
	Bits 2-0 = 0	(Réservés)
09h	Réservé	
0Ch	Cylindre de Landing Zone	
0Eh	Nombre de secteurs par piste	
0Fh	Réservé	

Tableau 4.11
Table des paramètres du disque dur (1 : PC-XT ; 2 : PC-AT).

Les AT contiennent aussi des données concernant les disques et les disquettes en RAM CMOS. Nous examinons ici leur format.

Données disques contenues en RAM CMOS

Déplacement	*Description*
0Eh	Diagnostic : Bit 3 = 1 Erreur d'initialisation du disque dur ou de son contrôleur Bit 1 = 1 Le contrôleur ne correspond pas à la configuration
11h	Type du disque n° 0
12h	Type du disque n° 1
19h	Type du lecteur de disque n° 1
1Ah	Type du lecteur de disque n° 2

Tableau 4.12
Données en RAM CMOS concernant les disques durs.

Le programme exemple concernant le format des disques durs est semblable à celui concernant les disquettes. Il est cependant plus simple et plus riche : il donne les valeurs entreposées en RAM CMOS. La clarté relative de ce programme tient essentiellement à la structure des informations fournies par le BIOS sur le disque dur, beaucoup plus cohérente que celles concernant les disquettes.

Listing 4.13
Programme ParamDsk.Pas. ❶

```
PROGRAM TableParametresDisques;        [ParamDsk.Pas]

{$R+}

USES Dos, Crt, Sys;

TYPE Table = ARRAY[0..$0F] OF Byte;
     Affic = ARRAY[0..9] OF STRING[60];

VAR TableParam : Table;
    TableBios  : ARRAY[1..7] OF STRING[60];
    CmosTab    : ARRAY[1..7] OF STRING[60];
    Champs     : Affic;

FUNCTION SegTableParam : Word;
BEGIN
  SegTableParam := MemW[$0000:($41 SHL 2)+2];
END;
```

➡

Programme ParamDsk.Pas (suite). ❷

```
FUNCTION OfsTableParam : Word;
BEGIN
  OfsTableParam := MemW[$0000:($41 SHL 2)];
END;

PROCEDURE CmosRamDsk;
VAR OctetLu : Byte;
    Tempo   : STRING[3];

BEGIN
  CmosTab[1] := ' Octet de Diagnostic                          : ';
  CmosTab[2] := ' Bit 3 = Erreur d''initalisation du ';
  CmosTab[2] := CmosTab[2] + 'contrôleur  : ';
  CmosTab[3] := ' Bit 1 = Configuration Disque non ';
  CmosTab[3] := CmosTab[3] + 'reconnue : ';
  CmosTab[4] := ' Type du Disque dur n° 0                     : ';
  CmosTab[5] := ' Type du Disque dur n° 1                     : ';
  CmosTab[6] := ' Type du Lecteur de Disque n° 1              : ';
  CmosTab[7] := ' Type du Lecteur de Disque n° 2              : ';
  CLI;
  Port[$0070] := $0E;
  OctetLu := Port[$0071];
  CmosTab[1] := CmosTab[1] + OctetDecVersBin(OctetLu)+'b';
  Tempo := Boole((OctetLu AND $08) SHR 3);
  CmosTab[2] := CmosTab[2] + Tempo;
  Tempo := Boole((OctetLu AND $02) SHR 1);
  CmosTab[3] := CmosTab[3] + Tempo;
  Port[$0070] := $11;
  OctetLu := Port[$0071];
  Str(OctetLu, Tempo);
  CmosTab[4] := CmosTab[4] + Tempo;
  Port[$0070] := $12;
  OctetLu := Port[$0071];
  Str(OctetLu, Tempo);
  CmosTab[5] := CmosTab[5] + Tempo;
  Port[$0070] := $19;
  OctetLu  := Port[$0071];
  Str(OctetLu, Tempo);
  CmosTab[6] := CmosTab[6] + Tempo;
  Port[$0070] := $1A;
  OctetLu := Port[$0071];
  Str(OctetLu, Tempo);
  CmosTab[7] := CmosTab[7] + Tempo;
  STI;
END;
```

➥

Programme ParamDsk.Pas (suite). ❸

```
PROCEDURE InitChampsBios;
VAR i      : Integer;
    Tempo : STRING;

   FUNCTION Message(Mot : Word) : STRING;
   BEGIN
     CASE Mot OF
       0 : Message := ' (Pas d''erreur )';
       1 : Message := ' Fonction demandée Inexistante ';
       2 : Message := ' Adresse introuvable ';
       3 : Message := ' Erreur de protection en écriture ';
       4 : Message := ' Secteur introuvable ';
       5 : Message := ' Erreur à la réinitialisation ';
       7 : Message := ' Drive mal paramétré ';
       8 : Message := ' Opération DMA impossible ';
       9 : Message := ' Erreur de données ';
     $0A : Message := ' Mauvais secteur détecté ';
     $0B : Message := ' Mauvaise piste détectée ';
  $0D : Message := ' Nombre de secteurs à formater erroné ';
  $0E : Message := ' Mauvaise adresse de contrôle données ';
     $0F : Message := ' Niveau DMA hors des limites ';
     $10 : Message := ' Erreur CRC ou ECC incorrigible ';
  $11 : Message := ' Erreur en correction des données ECC ';
     $20 : Message := ' Erreur générale du contrôleur ';
     $40 : Message := ' Erreur lors d''une opération Seek ';
     $80 : Message := ' Time-out ';
     $AA : Message := ' Drive non prêt ';
     $BB : Message := ' Erreur indéfinie ';
     $CC : Message := ' Erreur d''écriture ';
     $E0 : Message := ' Registre des Entrées/Sorties à 0 ';
     $FF : Message := ' Erreur en opération "Sense" ';
     ELSE
       Message := '';
     END; {CASE}
END;

BEGIN                            {Init Champs BIOS}
  Tempo := ' Erreur n° ' +
             MotDecVersHex((MemW[$0040:$0074] AND $FF));
  TableBios[1] := Tempo +
                  Message((MemW[$0040:$0074] AND $FF));
  TableBios[3] := ' Octet de Contrôle                : ' +
                    OctetDecVersBin(Mem[$0040:$0076]) + 'b'
         +' (' + OctetDecVersHex(Mem[$0040:$0076]) + 'h)';
```

➥

Programme ParamDsk.Pas (suite). ❹

```
   TableBios[4] := ' Offset du port du Disque Fixe     :  '+
                    OctetDecVersHex(Mem[$0040:$0077]) + 'h';
   IF AT THEN
   BEGIN
     Str(Mem[$0040:$0075], Tempo);
     TableBios[2] := ' Nombre de Disques Fixes          : ' +
                      Tempo;
     TableBios[5] := ' État du contrôleur de Disque Fixe : '
                   + OctetDecVersHex(Mem[$0040:$008C]) + 'h';
     TableBios[6] := ' N° d''Erreur du contrôleur :        '
                  + OctetDecVersHex(Mem[$0040:$008D]) + 'h';
     TableBios[7] := ' Drapeau d''interruption    :       ' +
                    OctetDecVersHex(Mem[$0040:$008E]) + 'h';
   END
   ELSE
   BEGIN
     TableBios[2] := TableBios[3];
     TableBios[3] := TableBios[4];
     FOR i := 4 TO 7 DO
       TableBios[i] := '';
   END;
END;

PROCEDURE InitChampsParam;
VAR i      : Integer;
    Tempo  : STRING;

BEGIN
   Move(Mem[SegTableParam:OfsTableParam],
   Mem[Seg(TableParam):Ofs(TableParam)], $10);
   FOR i:=0 TO 9 DO
     Champs[i] := '';
   Champs[0] := ' Nombre de Cylindres          :     ';
   Champs[1] := ' Nombre de Têtes              :        ';
   Champs[3] := ' Cylindre de Précompensation :     ';
   i := ((TableParam[1] SHL 8) + (TableParam[0]));
   Str(i, Tempo);
   Champs[0] := Champs[0] + Tempo + 'd';
   Str(TableParam[2], Tempo);
   Champs[1] := Champs[1] + Tempo + 'd';
   Champs[3] := Champs[3] + OctetDecVersHex(TableParam[6]) +
                OctetDecVersHex(TableParam[5]) + 'h';
```

➥

Programme ParamDsk.Pas (suite). ❺

```
  IF AT THEN
  BEGIN
      Champs[2] := ' Réservé                        :     ' +
                    OctetDecVersHex(TableParam[4]) +
                    OctetDecVersHex(TableParam[3]) + 'h';
      Champs[4] := ' Réservé                        :     '
                    + OctetDecVersHex(TableParam[7]) + 'h';
      Champs[5] := ' Octet de Contrôle              : ' +
                    OctetDecVersBin(TableParam[8]) + 'b' +
               ' (' + OctetDecVersHex(TableParam[8]) + 'h)' ;
      Champs[6] := ' Réservé                        :    ' +
                     OctetDecVersHex(TableParam[$B]) +
                     OctetDecVersHex(TableParam[$A]) +
                     OctetDecVersHex(TableParam[9]) + 'h' ;
      Champs[7] := ' Landing Zone                   :     ' +
                    OctetDecVersHex(TableParam[$D]) +
                    OctetDecVersHex(TableParam[$C]) + 'h' ;
      Champs[8] := ' Secteurs Par Piste             :      ';
      Str(TableParam[$E], Tempo);
      Champs[8] := Champs[8] + Tempo + 'd';
      Champs[9] := ' Réservé                        :      ' +
                    OctetDecVersHex(TableParam[$F]) + 'h';
  END
  ELSE
  BEGIN
      Champs[2] := ' Cylindre courant                 :    ' +
                     OctetDecVersHex(TableParam[4]) +
                     OctetDecVersHex(TableParam[3]) + 'h';
      Champs[4] := ' ECC Maximum                      :    '+
                     OctetDecVersHex(TableParam[7]) + 'h';
      Champs[5] := ' Octet de Contrôle                : ' +
                     OctetDecVersBin(TableParam[8]) + 'b' +
                ' (' + OctetDecVersHex(TableParam[8]) + 'h)';
      Champs[6] := ' Valeur de Timeout                :     ' +
                    OctetDecVersHex(TableParam[9]) + 'h';
      Champs[7] := ' Valeur Timeout pour le Formatage : ' +
                    OctetDecVersHex(TableParam[$A]) + 'h';
      Champs[8] := ' Timeout pour la Vérification : ' +
                    OctetDecVersHex(TableParam[$B]) + 'h';
      Champs[9] := ' Réservé                         :    ' +
                          OctetDecVersHex(TableParam[$F]) +
                          OctetDecVersHex(TableParam[$E]) +
                          OctetDecVersHex(TableParam[$D]) +
                     OctetDecVersHex(TableParam[$C]) + 'h';
  END;
END;
```

➡

Programme ParamDsk.Pas (suite). ❻

```
PROCEDURE AfficheChamps;
VAR i : Integer;
BEGIN
  TextAttr:=15+4*16; GotoXy(24,5);
  Write(' Table de Paramètres en ',
  MotDecVersHex(SegTableParam), ':');
  Write(MotDecVersHex(OfsTableParam), ' ');
  Window(8, 6, 72, 17); TextAttr:=14+4*16; ClrScr; WriteLn;
  FOR i:=0 TO 9 DO
    WriteLn('':10,Champs[i]);
  ReadLn; ClrScr; Window(1,1,80,25);
  GotoXy(20,5); TextAttr:=15+1*16; Write('':52);
  GotoXy(32,5); TextAttr:=15+4*16;
  Write(' Données RAM BIOS');
  Window(8, 6, 72, 17); WriteLn; WriteLn;
  TextAttr := 14+4*16;
  {$R-}
  i := 1;
  REPEAT
    WriteLn(' ', TableBios[i]);
    Inc(i);
  UNTIL (i > 7) OR (TableBios[i] = '');
  {$R+}
  IF AT THEN
  BEGIN
    ReadLn; ClrScr; Window(1,1,80,25);
    GotoXy(30, 5); TextAttr := 15+1*16; Write('':52);
    GotoXy(32, 5); TextAttr := 15+4*16;
    Write(' Données CMOS RAM');
    Window(8,6,72,17); WriteLn; WriteLn;
    TextAttr := 14+4*16;
    FOR i := 1 TO 7 DO
    BEGIN
      WriteLn('':3, CmosTab[i]);
    END;
  END;
  ReadLn; Window(1,1,80,25); TextAttr:=15+1*16; ClrScr;
END;
BEGIN
  TextAttr:=15+1*16; ClrScr;
  InitChampsParam;
  InitChampsBios;
  IF AT THEN
    CmosRamDsk;
  AfficheChamps;
END.
```

Formater une disquette avec les fonctions du BIOS

Le programme de formatage physique de disquette qui suit est une application pratique de ce que l'on peut réaliser avec les informations vues dans ce chapitre. Bien qu'il ne permette pas d'utiliser directement les disquettes formatées avec son aide (il faudrait les formater aussi au niveau logique), ni de formater un disque dur, il illustre bien l'emploi des interruptions du BIOS par l'intermédiaire du Turbo Pascal.

Listing 4.14
Programme Format.Pas. ❶

```
PROGRAM FormatagePhysiqueDsktte;         {Format.Pas}

USES Crt, Dos;

VAR Dsk,
    Erreur,
    NbPistes,
    NbSect,
    NbTetes,
    NbDskt    : Byte;
    Code      : Integer;
    Format,
    SegParam,
    OfsParam : Word;
    NbP,
    NbS       : STRING;
    Verifie   : BOOLEAN;

PROCEDURE AnalyseCommande(VAR Verifie : BOOLEAN; VAR NbP,
                          NbS : STRING);
VAR  i              : Word;
    Chaine, SChaine : STRING;
BEGIN
 Chaine:=''; SChaine:='';
 FOR i:=1 TO ParamCount DO
 BEGIN
   SChaine:=ParamStr(i);
   Chaine:=Chaine+SChaine;
 END;
 i:=1;
```

➥

Programme Format.Pas (suite). ❷

```
REPEAT
   Chaine[i]:=UpCase(Chaine[i]);
   CASE Chaine[i] OF
     'V' : Verifie:=TRUE;
     'P' : NbP:=Copy(Chaine, i+2, 2);
     'S' : NbS:=Copy(Chaine, i+2, 2);
   END;
   Inc(i);
 UNTIL i > Length(Chaine);
 IF Not (NbS[2] In ['0'..'9']) THEN
   NbS:=Copy(NbS, 1, 1);
 Val(NbP, NbPistes, Code);
 Val(NbS, NbSect, Code);
 IF (NbPistes = 39) THEN
   IF (Format = 120) THEN
     Format:=360
   ELSE
   IF (Format = 144) THEN
     Format:=720;
END;

FUNCTION DriveAFormater : Byte;
VAR  i       : Integer;
     Lettre : STRING[1];
BEGIN
 Lettre:=ParamStr(1);
 Lettre[1]:=UpCase(Lettre[1]);
 DriveAFormater:=Ord(Lettre[1])-65;
END;

PROCEDURE LitParametresDsktte(Drive : Byte; VAR NbPistes,
                              NbSect, NbTetes, NbDskt :
                              Byte; VAR SegParam, OfsParam,
                              Format : Word);
VAR  Regs : Registers;

BEGIN
 WITH Regs DO
 BEGIN
   Ah:=$8;
   Dl:=Drive;
   Intr($13, Regs);
   IF (Flags And 1 = 0) THEN
```

➡

Programme Format.Pas (suite). ❸

```
    BEGIN
      CASE Bl OF
        1 : Format:=360;
        2 : Format:=120;
        3 : Format:=720;
        4 : Format:=144;
      END;
      NbPistes:=Ch;
      NbSect:=Cl;
      NbTetes:=Dh;
      NbDskt:=Dl;
      SegParam:=Es;
      OfsParam:=Di;
    END
    ELSE
      Write(#7);
  END;
END;

FUNCTION ConfDrivePourFormatage(Drive, NbPistes, NbSect :
                                Byte) : Byte;
VAR  Regs : Registers;
BEGIN
 WITH Regs DO
 BEGIN
   Ah:=$18;
   Ch:=NbPistes;
   Cl:=NbSect;
   Dl:=Drive;
   Intr($13, Regs);
   IF (Flags And 1 <> 0) THEN
     Write(#7);
   ConfDrivePourFormatage:=Ah;
 END;
END;

FUNCTION FormattePisteDsktte(Drive, NoPiste, NoTete,
                             NbSect : Byte; SegBuf,
                             OfsBuf : Word) : Byte;
VAR  Regs : Registers;
BEGIN
 WITH Regs DO
 BEGIN
   Ah:=$5;
   Al:=NbSect;
```

➥

Programme Format.Pas (suite). ❹

```
    Dh:=NoTete;
    Dl:=Drive;
    Ch:=NoPiste;
    Es:=SegBuf;
    Bx:=OfsBuf;
    Intr($13, Regs);
    IF (Flags And 1 <> 0) THEN
      Write(#7);
    FormattePisteDsktte:=Ah;
 END;
END;

FUNCTION VerifiePisteDsktte(Drive, NoSect, NoPiste,
                            NoTete, NbSect : Byte; SegBuf,
                            OfsBuf : Word) : Byte;
VAR  Regs : Registers;
BEGIN
 WITH Regs DO
 BEGIN
   Ah:=$4;
   Dh:=NoTete;
   Dl:=Drive;
   Ch:=NoPiste;
   Cl:=NoSect;
   Es:=SegBuf;
   Bx:=OfsBuf;
   Intr($13, Regs);
   VerifiePisteDsktte:=Ah;
 END;
END;

FUNCTION InitDsktte(Drive : Byte) : Byte;
VAR  Regs : Registers;
BEGIN
 WITH Regs DO
 BEGIN
   Ah:=$0;
   Dl:=Drive;
   Intr($13, Regs);
   InitDsktte:=Ah;
 END;
END;
```

➥

Programme Format.Pas (suite). ❺

```
FUNCTION InitialiseDsktte(Drive : Byte) : Byte;
VAR  i   : Integer;
     Err : ARRAY[0..3] OF Byte;
BEGIN
 Err[0]:=0;
 FOR i:=1 TO 3 DO
 BEGIN
   Err[i]:=InitDsktte(Drive);
   IF (Err[i] = 0) THEN
   BEGIN
     InitialiseDsktte:=0;
     Exit;
   END;
 END;
 FOR i:=1 TO 2 DO
   IF (Err[i] = Err[i]+1) THEN
     Inc(Err[0]);
 IF (Err[0] = 3) THEN
   InitialiseDsktte:=1
 ELSE
   InitialiseDsktte:=0;
END;

PROCEDURE VerifieDsktte(Drive, NoSecteur, Piste, Tete,
                        NbSect : Byte; SegParam, OfsParam :
                        Word);
VAR  Lig : Byte;
BEGIN
 Lig:=WhereY;
 GotoXy(1, Lig);
 Write('Vérifie  ');
 Erreur:=VerifiePisteDsktte(Drive, NbSect, Piste, Tete,
                            NbSect, SegParam, OfsParam);
 IF (Erreur <> 0) THEN
   Erreur:=InitialiseDsktte(Drive);
 IF (Erreur <> 0) THEN
 BEGIN
   GotoXy(5,24);
   Write('Erreur ', Erreur);
   Halt;
 END;
END;
```

➥

Programme Format.Pas (suite). ❻

```
PROCEDURE FormateDsktte(Drive, NbPistes, NbSect, NbTetes :
                        Byte; VAR Format, SegParam,
                        OfsParam : Word);
VAR  Tableau : ARRAY[1..18] OF RECORD
                                NoPiste,
                                NoTete,
                                NoSect,
                                TailleSect : Byte;
                              END;
    i, Piste, Tete : Integer;
    Erreur, Lig    : Byte;
BEGIN
 Lig:=WhereY+1;
 FOR i:=1 TO 18 DO
   Tableau[i].TailleSect:=Mem[SegParam:OfsParam+3];
 FOR Piste:=0 TO NbPistes DO
   FOR Tete:=0 TO NbTetes DO
   BEGIN
     FOR i:=1 TO 18 DO
     BEGIN
       Tableau[i].NoPiste:=Piste;
       Tableau[i].NoTete:=Tete;
       Tableau[i].NoSect:=i;
     END;
     GotoXy(1, Lig);
     Write('Formate    ');
     GotoXy(15,Lig);
     Write('Piste ', Piste:2, ' Tête ', Tete);
     Erreur:=FormattePisteDsktte(Drive, Piste, Tete,
                                 NbSect, Seg(Tableau),
                                 Ofs(Tableau));
     IF (Erreur <> 0) THEN
       Halt;
     IF Verifie THEN
       VerifieDsktte(Drive, NbSect, Piste, Tete, NbSect,
                     Seg(Tableau), Ofs(Tableau));
   END;
END;

BEGIN
 ClrScr;
 Verifie:=FALSE;
 Dsk:=DriveAFormater;
 LitParametresDsktte(Dsk, NbPistes, NbSect, NbTetes,
                     NbDskt, SegParam,OfsParam, Format);
```

➥

Programme Format.Pas (suite). ❼

```
Erreur:=ConfDrivePourFormatage(Dsk, NbPistes, NbSect);
 AnalyseCommande(Verifie, NbP, NbS);
 WriteLn('Format ', Format);
 WriteLn('Pistes ', NbPistes);
 WriteLn('Secteurs ', NbSect);
 WriteLn('Têtes ', NbTetes+1);
 WriteLn(NbDskt, ' Disquette(s)');
 WriteLn('Table de Paramètres en ',SegParam, ':',
         OfsParam);
 WriteLn;
 IF (Erreur <> 0) THEN
   Halt
 ELSE
   FormateDsktte(Dsk, NbPistes, NbSect, NbTetes, Format,
                 SegParam, OfsParam);
 WriteLn;
 WriteLn;
 Write('Disquette ', ParamStr(1), ' formatée ');
 WriteLn('physiquement. ');
 WriteLn;
END.
```

Conclusion

Nous avons passé en revue les diverses fonctions de l'Int 13h du BIOS ainsi que les informations concernant les disques et les disquettes qui se trouvent en RAM et en RAM CMOS.

Chapitre 5

Disques au niveau logique : le plan d'un disque

Mots-clefs

Cluster	Un cluster regroupe plusieurs secteurs (de 1 à 8 selon les formats de disque). Il s'agit d'une structure logique créée par le DOS.
Cylindres	Un cylindre regroupe plusieurs pistes de même rang. Sur une disquette, le cylindre 0 contient la piste 0 de la face 0 *et* la piste 0 de la face 1. Les cylindres, comme les pistes, sont numérotés à partir de 0. Un disque contient autant de cylindres *en tout*, qu'il comporte de *pistes par face*.
Faces	Un disque a deux faces par plateau. Les disques durs ayant au moins deux plateaux, ils ont un minimum de quatre faces. A chacune de ces faces correspond une tête de lecture/écriture.
Gap	Il y a deux types de gap. Les *gaps réels* sont l'espace qui se trouve entre chaque secteur d'une même piste. Les *faux gaps* contiennent un certain nombre d'informations utiles au contrôleur de disque(tte), et se trouvent au début de chaque secteur avant la zone de donnée.
Pistes	Chaque face d'un disque contient un certain nombre de pistes concentriques. Ces pistes contiennent les enregistrements de données effectués sur le disque.
Secteurs	Un secteur contient généralement 512 octets. C'est la structure de base en matière de disque. Les secteurs sont regroupés par piste.
Secteurs cachés	Les secteurs cachés d'un disque dur contiennent des informations vitales, telles que l'emplacement des secteurs défectueux, qui y ont été inscrites lors du formatage physique par le constructeur.
Secteurs réservés	Certains secteurs d'un disque sont dits "réservés". Ils contiennent des informations nécessaires à la gestion du disque par le BIOS et/ou le DOS.
Têtes de lecture / écriture	Il y a une tête de lecture/écriture par face d'un disque. Elles se déplacent ensemble latéralement (de piste en piste) et longitudinalement (de secteur en secteur).

Nous venons de voir comment fonctionnent les disques au niveau physique. Mais le système d'exploitation ne peut pas se contenter des caractéristiques physiques des disques : entre les cartes contrôleurs, les lecteurs et les différents types de disque, il ne s'y retrouverait absolument pas. Il lui faut donc organiser tout ceci logiquement.

Dans cette optique, le DOS doit structurer le matériel en respectant les contraintes qu'il lui impose.

Faces et pistes

Le chapitre précédent nous a appris la façon dont le disque était géré par le BIOS (tables de paramètres, données RAM et CMOS RAM). Toutefois, bien qu'il s'agisse de caractéristiques physiques, nous n'avons pas encore parlé des faces, des pistes et des secteurs. Comme le DOS a besoin de les connaître, il nous a semblé préférable d'examiner ces caractéristiques en même temps que l'organisation logique du matériel, étroitement liée à son organisation physique.

En théorie, un disque ou une disquette informatique ressemble à s'y méprendre à un disque musical 33 ou 45 tours (dans les faits, mieux vaut ne pas les confondre...). Les uns comme les autres ont deux faces par plateau (un disque dur étant composé de plusieurs plateaux, il a *au moins* quatre faces). Un disque sert à stocker des informations : il faut pouvoir le lire ou y écrire. Les têtes de lecture/écriture remplissent ce rôle en se déplaçant au dessus ou en dessous du disque. Contrairement au cas d'une platine musicale, il y a une tête de lecture/écriture par face. Pour pouvoir accéder à n'importe quelle face, un lecteur de disque n'a pas seulement besoin d'une tête de lecture/écriture par face du disque, mais également d'un dispositif de déplacement de ces têtes. Il n'y a guère que deux possibilités : soit chaque tête est indépendante, soit elles se déplacent ensemble ; c'est la seconde solution qui a été retenue, pour d'évidentes raisons de rapidité d'accès : il y a beaucoup moins de calculs à effectuer (voir *figure* page suivante).

Reprenant notre comparaison avec les disques musicaux, nous nous apercevons que chacune de leurs faces est divisée en sillons, tandis que chaque face d'un disque informatique contient plusieurs pistes. Tandis que les disques musicaux ne contiennent qu'un seul sillon inscrit en spirale, il y a plusieurs pistes concentriques sur chaque face d'un disque informatique. Ces différentes pistes servent à l'enregistrement des informations. Elles sont numérotées de 0 à x-1, la première piste étant située à l'extérieur du disque et la dernière piste près du centre. Bien qu'il y ait au moins deux faces par disque, les pistes de la face 0 sont numérotées exactement comme celles de la face 1, les pistes de même numéro se trouvant l'une en dessous de l'autre.

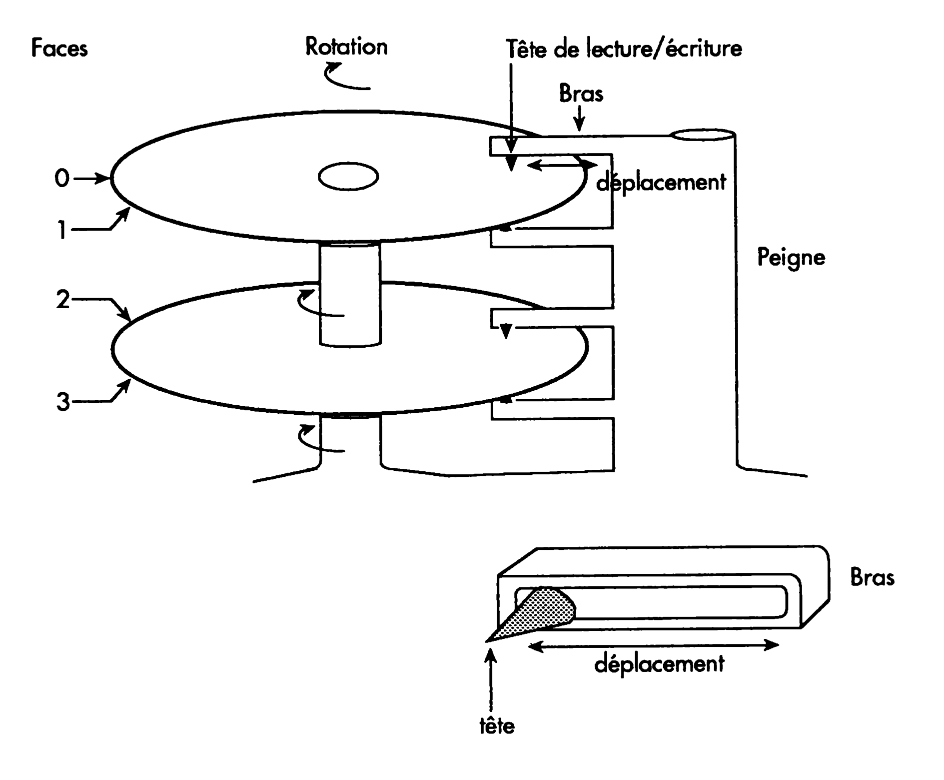


Figure 5.1
Faces et têtes d'un disque dur.

C'est pourquoi il faut non seulement préciser le numéro de la piste à laquelle on veut accéder, mais également celui de la face concernée. Ainsi, lorsque le BIOS veut écrire quelque part sur le disque, il transmet ces deux coordonnées au matériel. A partir de cette constatation, on a introduit la notion de cylindre.

Cylindres et secteurs

Un cylindre n'est pas autre chose que l'ensemble des pistes de même rang : le cylindre 0 d'une disquette contient la piste 0 de la face 0 et la piste 0 de la face 1 (voir *figure* 5.2). Le nombre total de cylindres correspond donc exactement au nombre de pistes par tête : une disquette 5 pouces 1/4 formatée en 360 Ko contient 40 cylindres numérotés de 0 à 39. Mais on indique tout de même le numéro de face lorsqu'on parle de cylindre : les différents programmes de formatage affichent à

l'écran "Face x, Cylindre y", puis "Face x+1, Cylindre y". Remplacez "Cylindre" par "Piste", et rien n'a changé !

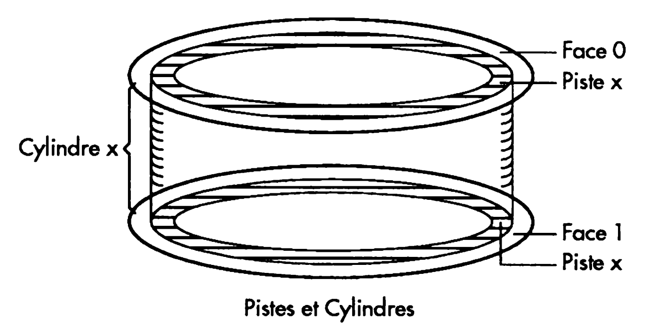


Figure 5.2
Pistes et cylindres.

Enfin, chaque piste est divisée en un certain nombre de secteurs. Là encore, deux stratégies coexistent : l'une consiste à inscrire un nombre de secteurs variable par piste, ce qui permet d'enregistrer plus d'informations mais augmente les temps d'accès (il y a automatiquement plus de calculs à effectuer pour que le bras se positionne sur le bon secteur). L'autre, la plus employée, inscrit le même nombre de secteurs sur chaque piste : on dispose donc de moins d'informations par disque, mais les temps d'accès sont réduits. Un secteur contient un certain nombre d'octets réservés à l'utilisateur (généralement 512). Le reste des octets, situé dans le *gap logique* , ou "faux gap", entre secteurs, est utilisé par le système pour inscrire des informations concernant la vérification des erreurs et le numéro de secteur. Pour accéder à un secteur, le matériel doit par conséquent connaître :

1. le numéro de la face ,
2. le numéro de la piste sur la face ,
3. le numéro du secteur dans la piste.

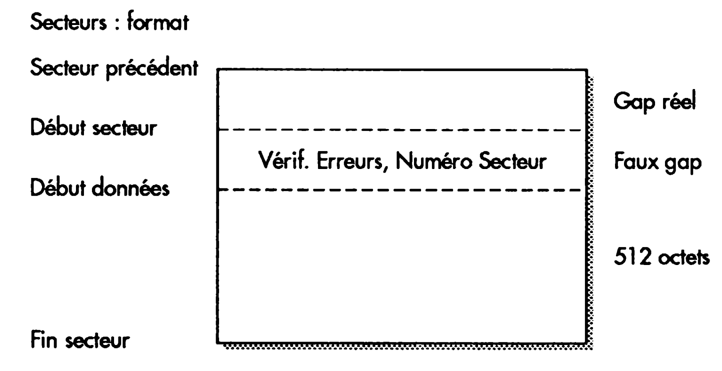


Figure 5.3
Format d'un secteur.

Ce schéma est valable pour tous les formats de disque. En revanche, les valeurs indiquées dans le *tableau 5.4* ne le sont que pour les disquettes :

Format	*Gap réel*	*Faux gap*
3 pouces 1/2	81 (51h) Octets	27 (1Bh) Octets
5 pouces 1/4	38 (26h) Octets	42 (2Ah) Octets

Tableau 5.4
Taille des gaps selon le format de disquette.

Ces valeurs sont susceptibles d'être modifiées. Elles sont cependant accessibles par l'intermédiaire de la table de paramètres des disquettes et sont calculées ainsi :

```
FauxGap:=Mem[Seg(TableParamDsktte):$05];

VraiGap:=(Mem[Seg(TableParamDsktte):$07]-FauxGap)
```

Les deux figures qui suivent, reproduites d'une documentation aimablement fournie par Western Digital, donnent le format exact d'un secteur de disquette (*figure 5.5*) et celui d'un secteur de disque dur (*figure 5.6*).

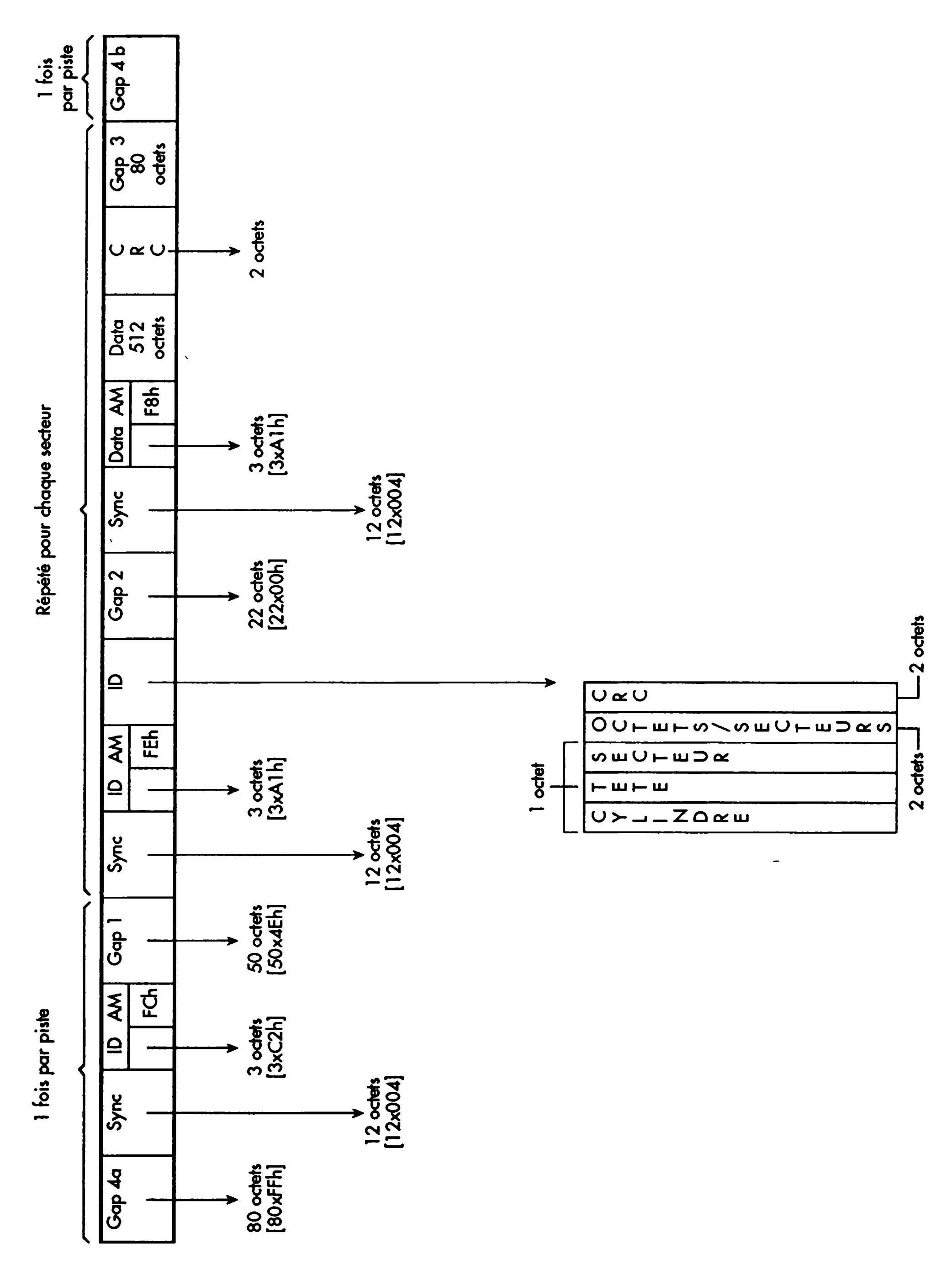


Figure 5.5
Format d'un secteur de disquette 5 pouces 1/4 lors d'un formatage MFM.

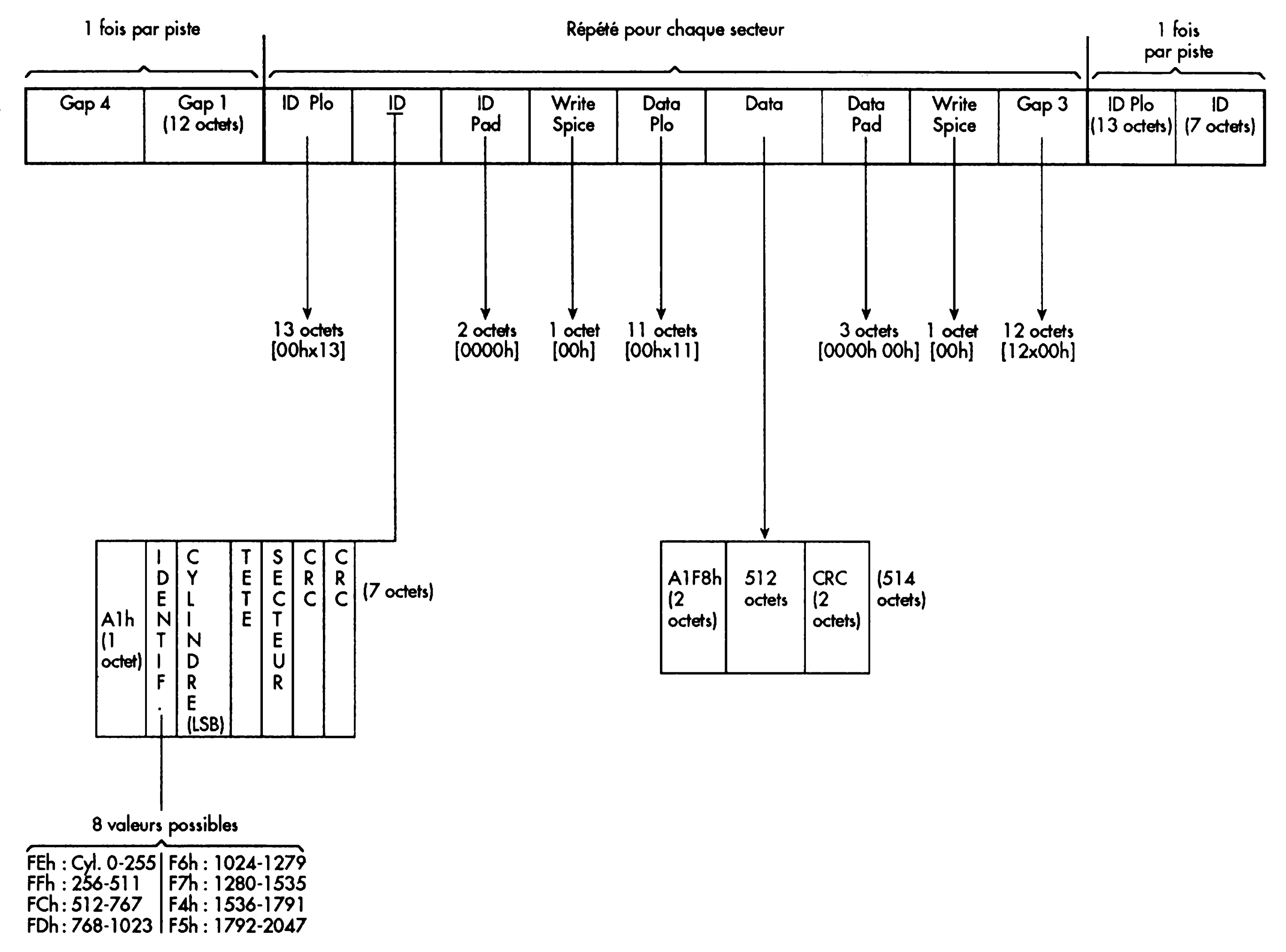


Figure 5.6
Format d'un secteur de disque dur lors d'un formatage MFM ou RLL.

Secteurs réservés et secteurs cachés

Le secteur 0 (face 0, piste 0, secteur 1) est repéré physiquement par un petit trou circulaire près du centre de la disquette, tous les autres numéros sont inscrits par logiciel au moment du formatage de la disquette. Sur un disque dur, le secteur 0 est indiqué différemment : une piste – parfois une face entière – est réservée aux informations concernant le disque. Ces informations se rapportent à l'emplacement de chaque piste et des secteurs physiques qu'elle contient. Elles sont inscrites sur le disque au moment du formatage de bas niveau effectué par le fabricant (certains utilitaires permettent d'y procéder soi-même). C'est ce qui fait qu'un disque dur a automatiquement un nombre impair de pistes (ou de faces) : la piste réservée n'est en effet jamais comptée. *PcShell* (Menu `Disk`, Option `Disk Info`) ou les *Norton Utilities* (programme `DI.EXE`) indiquent à l'auteur que son disque dur C: contient 17 secteurs par piste, 65 399 secteurs (donc 3 847 pistes), 17 secteurs cachés (donc une piste), 8 faces et 481 cylindres. Multipliez le nombre de cylindres par le nombre de têtes, vous obtiendrez le nombre réel de pistes que contient le disque : 3 848. On peut aussi additionner au nombre de pistes indiqué (3 847) le nombre de secteurs cachés (17) divisé par le nombre de secteurs par piste (17).

```
NbPisteReel := NbCylindre * NbTete                      (1)
NbPisteReel := NbPiste + (NbSectCache Div NbSectPP)     (2)
NbPisteReel := NbPiste + 1                              (3)
```

Légende

`NbPisteReel` :	Nombre réel de pistes
`NbCylindre` :	Nombre de cylindres (NbPiste / NbTete)
`NbPiste` :	Nombre de pistes indiqué
`NbTete` :	Nombre de têtes (ou de faces) du disque
`NbSectCache` :	Nombre de secteurs cachés
`NbSectPP` :	Nombre de secteurs par piste

La *formule 3* est la plus rapide pour déterminer le nombre réel de pistes du disque. Cependant, certains logiciels formatent à leur façon : ils peuvent aussi bien réserver une piste de plus que donner un nombre de secteurs par piste particulier, ou marquer certains secteurs comme mauvais. Comme le DOS doit pouvoir reconnaître ces disques, ces renseignements lui sont accessibles : c'est pourquoi nous conseillons plutôt l'emploi des *formules* 1 et 2.

Figure 5.7
Connaître le nombre exact de pistes d'un disque.

Cette piste cachée (à ne pas confondre avec les secteurs réservés) est indiquée par les programmes de mappage du disque les plus courants : l'écran du `Map Disk` de `PcShell` affiche un dernier petit carré occupé tout à la fin du disque. Si vous défragmentez votre disque à l'aide de `Compress` ou `Speed Disk` (pour *Norton Utilities*), vous vous apercevrez aisément que ce carré n'est jamais déplacé par le

logiciel, bien que n'étant jamais marqué non plus comme bloc fixe, comme mauvais bloc, ni même comme "bloc structure" (comme la FAT, le Boot Record ou encore le Root Directory).

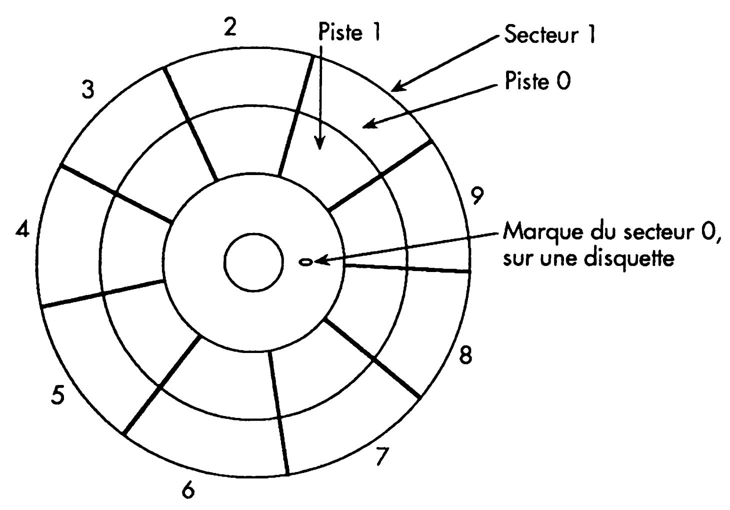


Figure 5.8
Secteur 0 et secteurs physiques sur disquettes.

On comprend mieux maintenant l'intérêt de ces rappels en matière d'organisation physique du disque. Le système d'exploitation est en effet obligé de tenir compte de cette structure. Comme il lui faut accéder à ce support (pour le formater, par exemple), il a évidemment besoin de connaître les subdivisions subtiles qui le régissent. Mais comme, d'un autre côté, l'utilisateur n'a pas à se préoccuper des numéros de face, de piste et de secteur contenant le fichier que son logiciel stocke pour lui, le système d'exploitation est bien obligé de lui présenter les choses sous un jour symbolique. D'autant que lui-même a besoin de ranger les fichiers sur le disque et de savoir où (dans quels secteurs) ils se trouvent...

Face à ces trois exigences, un système d'exploitation (et en tous les cas le DOS) procède en deux temps. Premièrement, il établit une correspondance entre secteurs physiques et secteurs logiques. Deuxièmement, il effectue des regroupements de secteurs dans une structure qui lui est propre.

Secteurs physiques et secteurs logiques

Faire correspondre secteurs physiques et secteurs logiques est indispensable : autant le DOS n'a pas à savoir où les données sont concrètement rangées par le matériel, autant il lui faut être en mesure de les retrouver et de contrôler leur bon emplacement. Pour cela, le DOS numérote les secteurs linéairement à partir de 0.

Chaque numéro de secteur logique correspond à un emplacement physique, calculé selon la formule encadrée.

```
SectLog:=(SectPhys-1)+(Tete*NbSPP)+(Pst*NbSPP*NbTete)
```

Légende

```
SectLog  = Numéro de Secteur logique ( >= 0)
SectPhys = Numéro de Secteur Physique ( >= 1)
Tete     = Numéro de Tête ( >= 0)
Pst      = Numéro de Piste ( >= 0)
NbSPP    = Nombre de Secteurs par Piste
NbTete   = Nombre de Têtes
```

Figure 5.9
Convertir les secteurs physiques en secteurs logiques.

Bien que la numérotation des secteurs logiques commence à `0`, les secteurs logiques du DOS ne sont accessibles qu'à partir de `1`. Le DOS n'utilisant que les numéros de secteurs logiques, on peut se demander à quel moment la conversion a lieu. En fait, ce n'est pas directement lui qui opère ce travail, mais le driver de périphériques disques. Celui-ci peut aussi bien être un driver spécifique à une marque de lecteur (il est alors déclaré dans le fichier `CONFIG.SYS`) que le driver par défaut, qui est chargé en mémoire par le fichier `IBMBIO.COM` au lancement du système. Le principe des secteurs logiques permet au DOS d'être certain que chaque requête d'écriture sur un secteur dont il a accepté le numéro correspond à un emplacement matériel du disque.

Clusters

Cependant, les secteurs contiennent généralement un nombre réduit d'octets par rapport à la longueur moyenne d'un fichier. Un autre problème, lié au précédent, provient du nombre relativement important de secteurs sur un disque : trop important pour que l'on puisse imaginer de garder la trace de chacun sans empiéter exagérément sur l'espace réservé aux données. En vue de résoudre ces deux difficultés, les concepteurs du DOS ont imaginé de regrouper les secteurs en *clusters* (ce que l'on traduit au choix par "grappes", "groupes", ou "blocs"...). Ceux-ci contiennent de 1 à 8 secteurs selon les formats de disques et sont donc facilement adressables. A taille de disque égale, plus un cluster contient de secteurs, et moins la structure chargée de les répertorier prend de place. Mais comme le DOS alloue de la place cluster par cluster, plus un cluster est grand et moins le disque peut contenir de fichiers.

Ce paradoxe apparent est facilement compréhensible : pour un disque de 720 secteurs de 512 octets chacun (cas typique d'une disquette de 360 Ko), si chaque cluster contenait 8 secteurs, le DOS n'aurait besoin que de 720 / 8 = 90 enregistrements de 8 bits pour en garder la trace, ce qui prendrait exactement

720 octets. Mais la position de chaque fichier écrit sur ce disque, aussi petit soit-il en réalité, devra être mémorisée dans un de ces 90 enregistrements : autrement dit, c'est un cluster entier (4 Ko) qui lui sera alloué. Le disque ne pourra donc contenir que 89 fichiers de moins de 4 Ko (un cluster entier étant réservé aux enregistrements eux-mêmes)... Comme on le voit, il est difficile de trouver un bon compromis : c'est pourquoi la taille des clusters varie en fonction de la taille des secteurs et de celle de l'espace disque adressable.

Format	*Taille*	*Faces*	*Pistes*	*Secteur*	*Cluster*
5 pouces 1/4	360 Ko	2	40	9	2 secteurs
	1.2 Mo	2	80	15	1 ....
3 pouces 1/2	720 Ko	2	80	9	2 ....
	1.44 Mo	2	80	18	1 ....
Disques	10 Mo	2	306	17	8 secteurs
	20 Mo	4	615	17	4 ....
	30 Mo	6	615	17	4.....
	46 Mo	6	940	17	4.....
	62 Mo	8	940	17	4.....
	112 Mo	15	900	17	4.....

Figure 5.10
Faces, pistes, secteurs, clusters : formats courants.

Les secteurs font tous 512 octets. Les chiffres donnés dans les rubriques pistes et secteurs s'entendent par rapport à la structure immédiatement au dessus (à droite dans le tableau) : il y a 40 pistes par face (en tout 80) et 9 secteurs par piste (360 par face, 720 en tout) sur une disquette 5 pouces 1/4 de 360 Ko. Les types de disques durs indiqués ne sont qu'un échantillon extrait d'une liste non exhaustive qui en répertorie 47. Ils ont été choisis parce que ce sont tous des disques IBM. Ceux qui disposent d'une table des types de disques durs auront reconnu les types n° 1, 2, 3, 5, 4 et 9. Enfin, toute partition de disque dur dont la taille est inférieure ou égale à 10 Mo contient des clusters de 8 secteurs (4 Ko), tandis que toute partition au-dessus de cette taille contient des clusters de 4 secteurs (2 Ko).

Connaître la structure d'un disque

Pour connaître la structure d'un disque, il existe deux moyens : utiliser l'Int `21h`, fonction `1Ch` ou accéder au secteur `0` (*Boot Record*, ou secteur de boot) par

l'intermédiaire de l'interruption 25h du DOS (lecture absolue sur disque d'un secteur logique). Chacune de ces techniques fournissant des renseignements différents, nous avons utilisé les deux. Mais comme il est impossible d'exécuter l'Int 25h par l'intermédiaire du Turbo Pascal, c'est en Assembleur que nous avons écrit ce premier programme-exemple.

DskStrct.Asm analyse le format d'un disque et fournit les renseignements suivants :

- Nombre de secteurs par cluster (Int 21h, Fct 1Ch)
- Nombre d'octets par secteur
- Nombre de clusters
- Identification OEM (Int 25h (BootRec))
- Nombre de secteurs réservés
- Nombre de secteurs par piste
- Nombre de têtes
- Nombre de secteurs cachés

L'Int 25h permet de lire le secteur logique 0 (secteur de boot), qui contient certaines informations indispensables. Nous en verrons la structure au chapitre suivant. Les renseignements qui nous intéressent sont situés aux déplacements :

- 03h à 0Bh : Identificateur OEM, chaîne de 8 caractères
- 0Eh à 10h : Secteurs réservés, mot
- 18h à 1Ah : Secteurs par piste, mot
- 1Ah à 1Ch : Têtes , mot
- 1Ch à 20h : Secteurs cachés, double mot

Si le nombre de secteurs réservés est à 1, alors il y a 0 secteur réservé (plus exactement, le seul secteur réservé est le secteur de boot lui-même).

Le programme lit la ligne de commande pour savoir quel disque est à analyser. Si elle est vide, DskStrct analysera le disque en cours. La longueur de la ligne de commande se trouve à l'offset 80h du PSP. En pointant sur le PSP, on effectue un MOV AX, ES:[0080h]. Si la longueur de la ligne de commande est supérieure à 0, on lit le second caractère (le premier est forcément un espace et le troisième le signe ":", qui ne nous intéresse pas), qu'il faut ensuite convertir en chiffre (1 pour A:, 2 pour B:, etc.).

`DskStrct` contient 8 procédures :

Nom de Procédures	Entrée	Int	Rôle
`Aff`	`Ax`	`21h, 02h`	Affiche un caractère
`HexaVersDec`	`Ax`	Aucune	Affiche un chiffre décimal Utilise `Aff`
`AffChaine`	`Dx, Bx`	`21h, 09h`	Affiche une chaîne
`DiagParSect`	`Dx, Bx`	`25h`	Lit le secteur de boot, affiche les données Appelle `HexaVersDec`, AffChaine
`DiagParInt`	`Dx, Bx`	`21h, 1Ch`	Affiche les données en provenance de l'Int. Appelle `HexaVersDec`, `AffChaine`
`Defaut`	`Bl`	`21h, 19h`	Lit le numéro de drive par défaut
`Drive`	`Bl`	Aucune	Transforme la lettre identifiant le drive en numéro (1 = `A:`, etc.)
`Fin`	Rien	`21h, 4Ch`	Termine

Tableau 5.11
Références croisées de DskStrct.Asm.

Listing 5.12
Programme DskStrct.Asm. ❶

```
.Model Small
.Data?
Tableau Db 512 Dup (?)
.Data
SpC Db ' Secteurs par Cluster', 0Ah, 0Dh, '$'
IdF Db ' Identification unité', 0Ah, 0Dh, '$'
OpS Db ' Octets par Secteur', 0Ah, 0Dh, '$'
NbC Db ' Clusters', 0Ah, 0Dh, '$'
SRv Db ' Secteurs réservés', 0Ah, 0Dh, '$'
Sc  Db ' Secteurs cachés', 0Ah, 0Dh, '$'
SpP Db ' Secteurs par Piste', 0Ah, 0Dh, '$'
NbT Db ' Têtes', 0Ah, 0Dh, '$'

.Stack 200h

.Code
```

➥

Programme DskStrct.Asm (suite). ❷

```
Aff           Proc     Near
              Public   Aff
      Push Ax             ; Sauver registres employés
      Push Dx
      Mov  Dl, Al         ; Al = Caractère à afficher,
                          ; Mettre dans Dl
      Mov  Ah, 02h        ; Afficher par INT 21h Fonction 02h
      Int  21h
      Pop  Dx             ; Restaurer les registres
      Pop  Ax
      Ret                 ; Fin de Procédure
Aff   EndP

HexaVersDec   Proc     Near
              Public   HexaVersDec

      Push Ax             ; Sauve les registres utilisés
      Push Cx
      Push Dx
      Push Si
      Xor  Cx, Cx         ; Mise à zéro de Cx
      Mov  Si, 0Ah        ; Divisions par 10

Conv:

      Inc  Cx             ; Cx sert de compteur
      Xor  Dx, Dx         ; Dx mis à zéro
      Div  Si             ; Dx:Ax / Si
      Push Dx             ; Sauver le reste de la division
      Cmp  Ax, 0000h      ; Fin ?
      Ja   Short Conv     ; Non, on boucle
@A:
      Pop  Ax             ; Restaurer le dernier reste
                          ; dans Ax
      Add  Ax, 30h        ; Rendre ASCII ('0'= #48 = #$30)
      Call Aff            ; Afficher
      Loop @A             ; Dec(Cx); If Cx > 0 Then @A

      Pop  Si             ; Restaurer registres
      Pop  Dx
      Pop  Cx
      Pop  Ax
      Ret                 ; Fin de Proc
HexaVersDec EndP
```

➥

Programme DskStrct.Asm (suite). ❸

```
AffChaine   Proc    Near
            Public  AffChaine

      Push Ax           ; Sauver Registres utilisés
      Push Dx
      Push Bx
      Mov  Ah, 09h      ; Fonction 09 de l'Int 21h
      Mov  Ds, Bx       ; Ds:=SEG Chaine et
      Int  21h          ; Dx:=OFFSET Chaine
      Pop  Bx           ; Afficher
      Pop  Dx           ; Restaurer registres
      Pop  Ax
      Ret               ; Fin Proc
AffChaine   EndP

DiagParSect Proc    Near
            Public  DiagParSect
      Push Ax          ; Sauve les Registres utilisés
      Push Dx
      Push Bx
      Push Cx
      Push Si
      Mov  Al, Bl               ; Al:=N° de drive
      Mov  Cx, 0001h            ; Lire 1 seul secteur
      Mov  Dx, 0000h            ; Secteur n° 0
      Mov  Bx, SEG Tableau      ; Tableau contiendra
      Mov  Ds, Bx               ; les octets lus
      Mov  Bx, OFFSET Tableau
      Int  25h         ; Lecture absolue sur disque
      Add  Sp, 2       ; A la sortie de l'Int, les
                       ; flags restent sur la pile :
                       ; il faut donc les enlever
      Mov  Si, 0000h   ; Mettre SI à 0
      Mov  Cx, 0008h   ; 8 caractères à afficher

BcleId:
      Mov  Ax, Ds:[Bx+03h+Si] ; caractère en Ax
      Call Aff                ; Afficher
      Inc  Si                 ; Pointe sur le caractère suivant
      Loop BcleId             ; jusqu'au 8° caractère
      Push Bx                 ; Sauver Bx, on va s'en servir
      Mov  Dx, OFFSET IdF     ; Bx:Dx := Chaine
      Mov  Bx, SEG IdF
      Call AffChaine          ; Afficher la chaîne
```

➥

Programme DskStrct.Asm (suite). ❹

```
      Pop  Bx                  ; Récupérer Bx pour l'adressage
      Mov  Ax, Ds:[Bx+0Eh]     ; Ax:= secteurs réservés
      Cmp  Ax, 0001h           ; Si = 1 Alors 0 secteur réservé
      Je   Short MoinsUn
      Jmp  Short Reste         ; Si <> 1 poursuivre
MoinsUn:
      Dec  Ax
Reste:
      Call HexaVersDec         ; Afficher
      Push Bx
      Mov  Dx, OFFSET SRv
      Mov  Bx, SEG SRv
      Call AffChaine
      Pop  Bx
      Mov  Ax, Ds:[Bx+18h] ; Nombre de secteurs par piste
      Call HexaVersDec
      Push Bx
      Mov  Dx, OFFSET SpP
      Mov  Bx, SEG SpP
      Call AffChaine
      Pop  Bx
      Mov  Ax, Ds:[Bx+1Ah] ; Nombre de têtes
      Call HexaVersDec
      Push Bx
      Mov  Dx, OFFSET NbT
      Mov  Bx, SEG NbT
      Call AffChaine
      Pop  Bx
      Mov  Ax, Ds:[Bx+1Dh] ; Lire d'abord le mot
      Cmp  Ax, 0000h       ; de poids fort, si=0
      Je   Short Second    ; ne pas afficher
      Call HexaVersDec
Second:
      Mov  Ax, Ds:[Bx+1Ch] ; Mot de poids faible
      Call HexaVersDec     ; Secteurs cachés
      Mov  Dx, OFFSET Sc
      Mov  Bx, SEG Sc
      Call AffChaine
      Pop  Si              ; Restaurer registres
      Pop  Cx
      Pop  Bx
      Pop  Dx
      Pop  Ax
      Ret                  ; Fin de procédure
DiagParSect EndP
```

➥

Programme DskStrct.Asm (suite). ❺

```
DiagParInt  Proc    Near
            Public  DiagParInt

      Push Ax             ; Sauve les registres employés
      Push Dx
      Mov  Ah, 1Ch        ; code fonction 1Ch
      Mov  Dl, Bl         ; Dl:= drive (0 = défaut, 1 = A:)
      Int  21h            ; Exécuter l'Int
      Push Dx             ; Sauver Dx (nombre de clusters),
                          ; on va s'en servir
      Xor  Ah, Ah         ; Ah := 0 (secteurs/ clusters dans Al)
      Call HexaVersDec             ; Afficher secteurs par cluster
      Mov  Dx, OFFSET SpC          ; Chaîne dans Bx:Dx
      Mov  Bx, SEG SpC
      Call AffChaine        ; Afficher Chaine
      Mov  Ax, Cx           ; Ax := octets par secteur
      Call HexaVersDec
      Mov  Dx, OFFSET OpS
      Mov  Bx, SEG OpS
      Call AffChaine
      Pop  Dx               ; Récupérer Dx
      Mov  Ax, Dx           ; Ax := Nombre de clusters
      Call HexaVersDec
      Mov  Dx, OFFSET NbC
      Mov  Bx, SEG NbC
      Call AffChaine
      Pop  Dx               ; Restaurer registres
      Pop  Ax
      Ret                   ; Fin de procédure
DiagParInt  EndP

Defaut      Proc    Near
            Public  Defaut
      Push Ax             ; Sauver registre utilisé
      Mov  Ah, 19h        ; Int 21h, Fonction 19h
      Int  21h
      Mov  Bl, Al         ; Bl := numéro drive
      Pop  Ax             ; Restaurer registre utilisé
      Ret                 ; Fin Proc
Defaut      EndP

Drive       Proc    Near
            Public  Drive
      Sub  Bl, 40h        ; 41h = 'A', 1 = drive A:
```

➥

Programme DskStrct.Asm (suite). ❻

```
      Cmp  Bl, 21h         ; 61h = 'a'
      Jnb  Short Minus     ; Si 'a' minuscule,  soustraire
      Jmp  Short Suite     ; sinon terminer
Minus:
      Sub  Bl, 20h
Suite:
      Ret                  ; Fin de Proc
Drive         EndP

Fin           Proc    FAR
              Public  Fin
      Mov  Ax, 4C00h       ; fonction 4Ch, ErrorLevel 0
      Int  21h
      RetF                 ; Fin Proc
Fin           EndP

Debut:
      Mov  Ax, @Data
      Mov  Ds, Ax            ; Segment de données en DS
      Mov  Al, Es:[0080h]    ; Longueur ligne de commandes
      Cmp  Al, 00h
      Je   Short Prg
      Mov  Bl, Es:[0082h]    ; Lettre identifiant
                             ; le drive dans BL
      Jmp  Short Milieu
Prg:
      Call Defaut            ; Si ligne de commande vide,
                             ; identifier le drive courant
      Jmp  Short Apres
Milieu:
      Call Drive             ; Quel lecteur doit-on analyser ?
Apres:
      Push Bx                ; Sauver numéro de drive
      Call DiagParInt        ; 1ère partie du programme
      Pop  Bx                ; Restaurer numéro de drive
      Cmp  Bl, 00h           ; Si drive = 0, Alors Ok
      Je   Short FinPrg
      Dec  Bl                ; Si drive > 0, Dec(drive)
                             ; (Dans la suite 0 = A:, 1 = B:
                             ;  avant, 1 = A:)
FinPrg:
      Call DiagParSect       ; Suite et fin du programme
      Call Fin
      End  Debut
```

Qu'y-a-t-il d'important dans ce programme ? Tout d'abord, la manipulation des paramètres en ligne de commande par le biais du registre ES. Ensuite, les deux principaux appels d'interruption DOS : l'Int 25h, qui permet de lire un secteur logique absolu sur disque, et la fonction 1Ch de l'Int 21h, qui donne certains renseignements sur le disque. Nous avons lu le secteur 0 du disque par l'Int 25h : ce secteur (le *Boot Record*) contient plusieurs précieux renseignements, dont certains n'ont pas été exploités ici. Nous verrons sa structure exacte au chapitre suivant.

Nous l'avons dit, le Turbo Pascal ne permet pas d'effectuer une lecture ou une écriture absolue sur disque. C'est la raison pour laquelle nous avons programmé en Assembleur. Toutefois, il serait inconcevable de programmer en Assembleur à chaque fois qu'un tel cas se présente. Nous allons présenter maintenant deux procédures que nous interfacerons ensuite avec nos programmes Pascal.

Listing 5.13
Programme Absolute.Asm. ❶

```
;****************** Absolute.Asm *******************;
;
; Procedure LitSectAbs(NoLect,NoSect,NbSect,@Tableau);
;
; NoLect    Byte       No du lecteur (A: = 0)
; NoSect    Word       No du premier secteur à lire
; NbSect    Word       Nombre de secteurs à lire
; @Tableau ^Byte       Pointeur sur le premier octet du
;                      tableau
; La procédure LitSectAbs renvoie les octets contenus
; dans les secteurs lus dans le tableau pointé.
;***************************************************;

Code   SEGMENT
       ASSUME Cs:Code

Tab       EQU  DWORD PTR Ss:[Bp+06]
NbSect    EQU  WORD  PTR Ss:[Bp+10]
NoSect    EQU  WORD  PTR Ss:[Bp+12]
NoLect    EQU  BYTE  PTR Ss:[Bp+14]

LitSectAbs    PROC   FAR ; appelable d'un TPU ou d'un programme
              PUBLIC LitSectAbs

        Push Bp
        Mov  Bp, Sp       ; Sauve la pile

        LDs  Si, Tab     ; Tableau en DS:SI
        Mov  Bx, Si      ; Bx := Ofs(Tableau)
        Mov  Cx, NbSect ; Cx := Nbre de secteurs à lire
```

➥

Programme Absolute.Asm (suite). ❷

```
        Mov  Dx, NoSect ; Dx := No du premier secteur
        Mov  Al, NoLect ; Al := No de lecteur

        Push Ds         ; Sauve les registres (détruits
        Push Bx         ; par l'Int)
        Push Cx
        Push Dx
        Push Ax
        PushF

        Int  25h
        Add  Sp, 2      ; L'int 25h fait un PushF, mais
                        ; aucun PopF en sortie
        PopF            ; Restaure les registres sauvés
        Pop  Ax
        Pop  Dx
        Pop  Cx
        Pop  Bx
        Pop  Ds
        Pop  Bp        ; Restaure la pile
        Ret  10        ; 10 octets ont été utilisés
LitSectAbs   EndP      ; Fin de procédure

;*********************************************************;
; Procedure EcritSectAbs(NoLect,NoSect,NbSect,@Tableau);
; Les paramètres sont les mêmes que ceux de LitSectAbs
; La procédure écrit sur disque les octets contenus dans
; le tableau dans des secteurs consécutifs
;*********************************************************;

EcritSectAbs PROC   FAR
             PUBLIC EcritSectAbs

        Push Bp
        Mov  Bp, Sp

        LDs  Si, Tab
        Mov  Bx, Si
        Mov  Cx, NbSect
        Mov  Dx, NoSect
        Mov  Al, NoLect
        Push Ds
        Push Bx
        Push Cx
```

➥

Programme Absolute.Asm (suite). ❸

```
        Push Dx
        Push Ax
        PushF

        Int  26h
        Add  Sp, 2

        PopF
        Pop  Ax
        Pop  Dx
        Pop  Cx
        Pop  Bx
        Pop  Ds

        Pop  Bp
        Ret  10
EcritSectAbs EndP

Code         EndS
             End
```

Une fois ce fichier assemblé, il reste à le déclarer avec les procédures qui le constituent dans le programme Pascal qui l'utilisera. Les deux procédures étant de type FAR (appel long), il est possible de s'en servir aussi bien dans un programme que dans une unité.

Le programme qui suit n'est qu'une illustration de l'interfaçage entre Assembleur et Pascal et ne nécessite pas d'autres commentaires que ceux qu'il contient. La seule bizarrerie qui apparaît tient à l'adresse de l'identificateur système (OEM) que l'on affiche. La boucle chargée de l'afficher est indicée à partir de 4, alors qu'il se trouve au déplacement 3. Cela est dû au fait que l'index du tableau commence à 1. S'il s'agissait d'un ARRAY[0..511] OF BYTE, l'index de la boucle FOR aurait coïncidé avec l'adresse de déplacement.

Listing 5.14
Programme Int25h.Pas. ❶

```
PROGRAM EssaiInt25EtInt26; { Int25h.Pas }

TYPE   BytePtr = ^BYTE;
VAR    Tableau : ARRAY[1..512] OF BYTE;
       i       : BYTE;
       IDS     : STRING[8];

{$L D:\Tasm\Absolute.Obj}  { Charge le Fichier Objet    }
{$F+}                      { Les Proc Asm sont FAR      }
```

Programme Int25h.Pas (suite). ❷

```
PROCEDURE LitSectAbs(NoLect: BYTE; NoSect,NbSect:WORD;
                   Tab : BytePtr); EXTERNAL;
PROCEDURE EcritSectAbs(NoLect: BYTE; NoSect,NbSect:WORD;
                      Tab : BytePtr); EXTERNAL;

{$F-}                       { Le reste du Programme est }
                            { NEAR par défaut           }

          { Disque D:, Secteur 0, Pointeur sur Tableau }
BEGIN
  LitSectAbs(3,0,1,@Tableau);
  Write('ID Système : ');
  FOR i:=4 TO 11 DO             { Id est à l'Offset 3 }
    Write(Chr(Tableau[i]));
  WriteLn;
  Write('Nouvelle ID : ');
  ReadLn(IDS);
  IF (IDS <> '') THEN
  BEGIN
    FOR i:=4 TO 11 DO       { Ecriture sur le disque de   }
      Tableau[i]:=Ord(IDS[i-3]);        { la nouvelle Id }
    EcritSectAbs(3,0,1,@Tableau);
  END
  ELSE
  BEGIN
     Write('L'opération d'écriture ne présente');
     WriteLn(' aucun danger');
  END;
                             { Relecture pour vérifier }
  LitSectAbs(3,0,1,@Tableau);
  Write('Nouvelle ID Système est bien : ');
  FOR i:=4 TO 11 DO
    Write(Chr(Tableau[i]));
END.
```

Afficher et modifier le contenu des secteurs

Maintenant que les principes ont été expliqués, nous allons passer à la pratique avec un programme de Dump secteur par secteur. Ce programme, *Dumper*, fait appel aux interruptions 25h et 26h pour pouvoir lire et écrire sur disque les secteurs dumpés (modifiables par l'utilisateur).

C'est naturellement le *Memory Dumper* du chapitre 3 qui a été remanié. Les principaux changements sont des remplacements de procédures par d'autres de même nature : ainsi, puisque les 512 octets affichés ne proviennent plus de la RAM mais d'un des secteurs logiques du disque, nous n'avons pas besoin de connaître la taille de la mémoire, mais plutôt le numéro du disque courant. La structure même du programme a donc subi peu de modifications, et il ne devrait pas être difficile de les repérer à l'aide du tableau qui suit.

Procédures internes à DUMPER :

Type	*Nom*	*Paramètres*	*Fonction*
P	Init	NoDsk (BYTE)	Initialise 3 variables globales
F	DskCourant		Renvoie le numéro de disque
F	NbDsk	No (BYTE)	Renvoie le nombre de disques et change de disque courant
P	Cadre	Col1, Lig1, Col2, Lig2 (BYTE)	Trace un cadre double aux couleurs en cours
P	Ecran		Trace l'écran principal
P	Menu		Affiche le menu
P	Barre	Octet (Fntr)	Affiche la barre d'informations
P	DumpSecteur	Drive (BYTE) VAR NoSect (WORD)	Dumpe un secteur (Hex)
P	DumpAsc		Dumpe un secteur (ASCII)
P	AffCurs	Col, Lig, Attr (BYTE), Octet (Fntr)	Affiche le curseur (ou l'efface selon Attr.)
P	Affiche	Lig,Indice (BYTE)	Affiche 1seize lignes du tableau de chaînes
P	AvertitEtSauve	NoDsk (BYTE) NoSect (WORD)	Envoie un message et sauve le secteur si Oui

➥

(suite du tableau)

Type	Nom	Paramètres	Fonction
P	PgeUp	VAR Lig,	Affiche la page précédente (suivante) ou le secteur selon les cas
P	PgeDn	Indice (BYTE) Octet (Fntr) VAR Ok (BOOLEAN)	
P	LitCar	Indice (BYTE)	Lit le clavier, modifie le secteur
P	Programme DumpSecteur		Appelle Menu, Init,

Tableau 5.15
Références croisées de Dumper.Pas.

Listing 5.16
Programme Dumper.Pas. ❶

```
PROGRAM Dumper;    { Dumper.Pas }
USES  Dos, Crt, Sys;

TYPE
  Sect      = ARRAY[1..512] OF BYTE;
  DumpSect  = ARRAY[1..32] OF STRING[80];
  Fntr      = RECORD
                NoOctet : WORD;
                Octet   : BYTE;
                Hexa    : STRING[2];
                Ascii   : CHAR;
              END;
  PntSect   = ^BYTE;

VAR
  Secteur               : Sect;
  SectDump              : DumpSect;
  Essai, NoDsk          : BYTE;
  NoSecteur, MaxSect,
  SectFat, SectRoot     : WORD;
  Chaine                : STRING;

{$L D:\PASCAL\SOURCES\ABSREAD.OBJ}
{$F+}
PROCEDURE LitSectAbs(NoLect: BYTE; NoSect,NbSect: WORD;
                     Tab: PntSect); EXTERNAL;
```

➥

Programme Dumper.Pas (suite). ❷

```
PROCEDURE EcritSectAbs(NoLect: BYTE; NoSect,NbSect: WORD;
                       Tab: PntSect); EXTERNAL;
{$F-}

PROCEDURE Init(NoDsk : BYTE);
VAR NbFat : BYTE;
BEGIN
  LitSectAbs(NoDsk, 0, 1, @Secteur);
  NbFat:=Secteur[$11];
  SectFat:=NbFat*((Secteur[$18] SHL 8)+Secteur[$17]);
  SectRoot:=((Secteur[$13] SHL 8)+Secteur[$12]) DIV 512;
  SectRoot:=(32*SectRoot)+SectFat;
  MaxSect:=(Secteur[$15] SHL 8)+Secteur[$14];
END;

FUNCTION DskCourant : BYTE;
VAR Regs : Registers;
BEGIN
  Regs.Ah:=$19;
  MsDos(Regs);
  DskCourant:=Regs.Al;
END;

FUNCTION NbDsk(No : BYTE) : BYTE;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah:=$0E;
    Dl:=No;
    MsDos(Regs);
    NbDsk:=Al
  END;
END;

PROCEDURE Cadre(Col1, Lig1, Col2, Lig2 : BYTE);
VAR i : INTEGER;
BEGIN
  FOR i:=Lig1 TO Lig2-1 DO
  BEGIN
    GotoXy(Col1, i);
    Write(#186, '':(Col2-Col1), #186);
  END;
  FOR i:=Col1+1 TO Col2 DO
```

➡

Programme Dumper.Pas (suite). ❸

```
  BEGIN
    GotoXy(i,Lig1-1); Write(#205);
    GotoXy(i,Lig2); Write(#205);
  END;
  GotoXy(Col1,Lig1-1); Write(#201);
  GotoXy(Col2+1,Lig1-1); Write(#187);
  GotoXy(Col1,Lig2); Write(#200);
  GotoXy(Col2+1,Lig2); Write(#188);
END;

PROCEDURE Ecran;
BEGIN
  TextAttr:=15+7*16; ClrScr;
  TextAttr:=14+1*16; Cadre(30,2,51,3);
  Cadre(2,7, 77,23); TextAttr:=14+4*16;
  GotoXy(2,24); Write('':77);
  GotoXy(32,2); Write(' D U M P E R  1.0 ');
END;

PROCEDURE Menu;
VAR Car : CHAR;
    No  : WORD;
BEGIN
  TextAttr:=15+7*16; Cadre(25,9, 55,20);
  GotoXy(30,11); Write(NbDsk(DskCourant),' Lecteurs ');
  GotoXy(30,13);
  Write(MaxSect,' Secteurs sur ',Chr(DskCourant+65),':');
  GotoXy(30,15); Write('Lecteur :  ');
  GotoXy(30,17); Write('Secteur :  ');
  GotoXy(41,15); Car:=ReadKey; Car:=UpCase(Car);
  IF (Car = #27) THEN
  BEGIN
    ClrScr;
    Halt;
  END;
  Write(Car,':');
  NoDsk:=Ord(Car)-65;
  No:=NbDsk(NoDsk);
  GotoXy(41,17); Read(No); NoSecteur:=No;
END;

PROCEDURE Barre(Octet : Fntr);
VAR Chaine : STRING;
```

➥

Programme Dumper.Pas (suite). ❹

```
BEGIN
  TextAttr:=15+7*16;
  GotoXy(1,5); Write('':80);
  TextAttr:=14+4*16;
  IF (NoSecteur = 0) THEN
    Chaine:=' Boot RECORD '
  ELSE
  IF (NoSecteur <= SectFat) THEN
    Chaine:=' FAT '
  ELSE
  IF (NoSecteur <= SectRoot) THEN
    Chaine:=' Root Directory '
  ELSE
    Chaine:=' Zone Données ';
  GotoXy(40-(Length(Chaine) DIV 2), 5); Write(Chaine);
  GotoXy(3,24);
  WITH Octet DO
  BEGIN
    Write(' Disque ', Chr(NoDsk+65), ': | ', ' Secteur');
    Write(NoSecteur:5,' | Hexa ',Hexa,' | Ascii ',Ascii);
    Write(' | Octet ',Octet:3,' | N° ',NoOctet:3);
  END;
END;

FUNCTION OctetHexVersDec(Octet : STRING) : BYTE;
VAR i  : INTEGER;
    No : BYTE;
BEGIN
  No:=0;
  FOR i:=1 TO 2 DO
    IF (Ord(Octet[i])-48 > 9) THEN
      No:=(No SHL 4) + Ord(Octet[i])-55
    ELSE
      No:=(No SHL 4) + Ord(Octet[i])-48;
  OctetHexVersDec:=No;
END;

PROCEDURE DumpSecteur(Drive : BYTE; VAR NoSect : WORD);
VAR i, k          : INTEGER;
    Chaine, Octet: STRING;

  PROCEDURE DumpAsc;
  VAR j : INTEGER;
  BEGIN
    Chaine:=Chaine+' | ';
```

➥

Programme Dumper.Pas (suite). ❺

```
    j:=(i-16);
    REPEAT
      IF NOT (Secteur[j] In [0..14]) THEN
        Chaine:=Chaine+Chr(Secteur[j])
      ELSE
        Chaine:=Chaine+' ';
      Inc(j);
    UNTIL (j > i-1);
    SectDump[k]:=SectDump[k]+Chaine; Chaine:='';
    Inc(k);
  END;

BEGIN
  Chaine:='';
  LitSectAbs(Drive, NoSect, 1, @Secteur);
  FOR i:=1 TO 32 DO
    SectDump[i]:=OctetDecVersHex(i-1)+' | ';
  i:=1; k:=1;
  REPEAT
    IF (Length(Chaine) = 48) THEN
      DumpAsc
    ELSE
    IF (Length(Chaine) < 48) THEN
    BEGIN
      Chaine:=Chaine+OctetDecVersHex(Secteur[i])+' ';
      Inc(i);
     END
  UNTIL (i > 512);
  DumpAsc;
END;

PROCEDURE AffCurs(Col,Lig,Attr: BYTE; Octet: Fntr);
VAR i : INTEGER;
BEGIN
  TextAttr:=Attr;
  GotoXy(Col, Lig); Write(Octet.Hexa);
  GotoXy(((Col-9) DIV 3)+60, Lig); Write(Octet.Ascii);
END;

PROCEDURE Affiche(Lig, Indice : BYTE);
VAR i : INTEGER;
BEGIN
  TextAttr:=15+1*16;
  i:=Indice;
```

➥

Programme Dumper.Pas (suite). ❻

```
    REPEAT
      GotoXy(4,Lig); WriteLn(SectDump[i+1]);
      Inc(i);
      Inc(Lig);
    UNTIL (i > Indice+15);
END;

PROCEDURE AvertitEtSauve(NoDsk : BYTE; NoSect : WORD);
VAR Car : CHAR;
BEGIN
  TextAttr:=14+4*16; Cadre(2,4,77,5);
  GotoXy(20,4);
 Write(' Sauver   Secteur n° ', NoSecteur, ' ? O/N : ');
  Car:=ReadKey; Car:=UpCase(Car); Write(Car);
  IF (Car = 'O') THEN
    EcritSectAbs(NoDsk, NoSect, 1, @Secteur);
  TextAttr:=7+7*16; Cadre(2,4,77,5);
END;

PROCEDURE PgeUp(VAR Lig,Indice: BYTE;Octet: Fntr;
                VAR Ok: BOOLEAN);
BEGIN
  IF (Octet.NoOctet > 256) THEN
  BEGIN
    Affiche(7,0);
    Indice:=0; Lig:=7;
  END
  ELSE
  IF (NoSecteur > 0) THEN
  BEGIN
    IF Ok THEN
      AvertitEtSauve(NoDsk, NoSecteur);
    Dec(NoSecteur);
    DumpSecteur(NoDsk, NoSecteur);
    Affiche(7,0);
    Indice:=0; Lig:=7; Ok:=False;
  END;
END;

PROCEDURE PgeDn(VAR Lig,Indice: BYTE;Octet: Fntr;
                VAR Ok: BOOLEAN);
BEGIN
  IF (Octet.NoOctet <= 256) THEN
  BEGIN
    Affiche(7, 16);
```

➥

Programme Dumper.Pas (suite). ❼

```
    Indice:=16; Lig:=7;
  END
  ELSE
  IF (NoSecteur < MaxSect) THEN
  BEGIN
    IF Ok THEN
      AvertitEtSauve(NoDsk, NoSecteur);
    Inc(NoSecteur);
    DumpSecteur(NoDsk, NoSecteur);
    Affiche(7, 0);
    Indice:=0; Lig:=7; Ok:=False;
  END;
END;

PROCEDURE LitCar(Indice: BYTE);
VAR Car                   : CHAR;
    Col, Lig, i, Compteur: BYTE;
    OctetEnCours          : Fntr;
    Modif                 : BOOLEAN;

  PROCEDURE Programme;
  BEGIN
    Menu;
    Init(NoDsk);
    DumpSecteur(NoDsk,NoSecteur);
  END;

BEGIN                   {LitCar}
  Programme;
  Car:=#215; Col:=9; Lig:=7;
  Compteur:=1; Modif:=False;
  Affiche(Lig, Indice);
  WITH OctetEnCours DO
  BEGIN
    NoOctet:=(Indice * 16)+(Col DIV 3)-2;
    Octet:=Secteur[NoOctet];
    Hexa:=SectDump[Indice+1][Col-3] +
    SectDump[Indice+1][Col-2];
    Ascii:=SectDump[Indice+1][((Col-9) DIV 3)+57];
  END;
  AffCurs(Col, Lig,(2*16+6), OctetEnCours);
  Barre(OctetEnCours);
  REPEAT
    Car:=ReadKey; Car:=UpCase(Car);
```

➥

Programme Dumper.Pas (suite). ❽

```
    CASE Car OF
    #0 : BEGIN
           Car:=ReadKey;
           AffCurs(Col,Lig,(1*16+15),OctetEnCours);
           CASE Car OF
           #73 : BEGIN
                   PgeUp(Lig,Indice,OctetEnCours,Modif);
                   Barre(OctetEnCours);
                 END;
           #77 : Inc(Col,3);
           #75 : Dec(Col,3);
           #72 : BEGIN
                   Dec(Lig);
                   Dec(Indice);
                 END;
           #80 : BEGIN
                   Inc(Lig);
                   Inc(Indice);
                 END;
           #81 : BEGIN
                   PgeDn(Lig,Indice,OctetEnCours,Modif);
                   Barre(OctetEnCours);
                 END;
           END;        {Case}
           IF (Col < 9) THEN
           BEGIN
             Col:=54;
             Dec(Lig);
             Dec(Indice);
           END
           ELSE
           IF (Col > 54) THEN
           BEGIN
             Col:=9;
             Inc(Lig);
             Inc(Indice);
           END;
           IF (Lig < 7) THEN
           BEGIN
             Lig:=22;
             Inc(Indice,16);
           END
           ELSE
           IF (Lig > 22) THEN
```

➥

Programme Dumper.Pas (suite). ❾

```
            BEGIN
              Lig:=7;
              Dec(Indice, 16);
            END;
            WITH OctetEnCours DO
            BEGIN
              NoOctet:=(Indice * 16)+(Col DIV 3)-2;
              Octet:=Secteur[NoOctet];
              Hexa:=SectDump[Indice+1][Col-3] +
              SectDump[Indice+1][Col-2];
              Ascii:=SectDump[Indice+1][((Col-9) DIV 3)+57];
            END;
            AffCurs(Col,Lig,(2*16+6),OctetEnCours);
          END;
     #65..#70,
     #48..#57 : BEGIN
                  WITH OctetEnCours DO
                  BEGIN
                    Hexa[Compteur]:=Car;
                    Octet:=HexaVersDecimal(Hexa);
                    Ascii:=Chr(Octet);
                    SectDump[Indice+1][Col-3]:=Hexa[1];
                    SectDump[Indice+1][Col-2]:=Hexa[2];
                    SectDump[Indice+1]
                    [((Col-9) DIV 3)+57]:=Ascii;
                    Secteur[NoOctet]:=Octet;
                  END;
                  AffCurs(Col,Lig,(2*16+6),OctetEnCours);
                  Modif:=True;
                  IF (Compteur < 2) THEN
                    Inc(Compteur)
                  ELSE
                    Compteur:=1;
                END;
     END;
     Barre(OctetEnCours);
   UNTIL (Car=#27);
   LitCar(0);
END;

BEGIN
 Init(DskCourant);
 Ecran;
 LitCar(0);
END.
```

Ce programme est long (environ 400 lignes), il est aussi performant et intéressant. Il permet en effet d'examiner un disque de fond en comble et de le modifier : secteur de boot, FAT, répertoire racine et zone fichier sont aisément accessibles et manipulables. Nous vous conseillons d'ailleurs de vous exercer d'abord sur des disquettes pleines dont vous aurez fait une copie au préalable. Lorsque vous maîtriserez bien les principes auxquels `Dumper` fait appel, vous pourrez le faire fonctionner sur disque dur.

Vous vous apercevrez vite qu'un disque s'organise selon différents schémas : le programme vous prévient de chaque changement de structure. En outre, la ligne d'informations située au bas de l'écran indique en permanence le disque examiné, le numéro du secteur, les valeurs hexadécimales, ASCII et décimales de l'octet situé sous le curseur, ainsi que son numéro d'offset relatif dans le secteur. On peut sans difficulté ajouter à ces renseignements le numéro absolu de déplacement de l'octet (numéro relatif * (numéro de secteur+1*512)), le numéro du cluster auquel ce secteur appartient (numéro de secteur+1 Div (nombre de secteurs par cluster)), et même les numéros de faces, de pistes et de secteurs physiques correspondant au secteur logique examiné...

Bien d'autres améliorations encore sont possibles et nous vous conseillons de les implémenter toutes. Quoiqu'il en soit, utilisez ce programme jusqu'à épuisement : vous apprendrez toujours quelque chose de nouveau sur les disques. Enfin, et c'est la dernière remarque à faire à son sujet, le code source du programme n'est pas commenté : c'est volontaire. Avec ce que nous avons dit jusqu'à présent sur la technique du dump et sur l'organisation du disque, cela n'est pas apparu nécessaire. Si un point restait malgré tout obscur, c'est à vous que reviendrait la charge de l'éclaircir. Sans quoi, où serait le charme de la programmation ?

Conclusion

Nous avons vu comment un disque était organisé physiquement et comment le DOS devait tenir compte de cette organisation à travers les notions de face, de piste, de cylindre, de secteur et de cluster. A partir de ces connaissances, nous avons écrit plusieurs programmes en Assembleur et en Turbo Pascal, dont un utilitaire de dump qui permet l'examen et la modification des secteurs d'un disque. Dans les deux chapitres suivants, nous allons étudier les structures logiques les plus proches du système d'exploitation : ce sont le secteur de boot, la FAT, le répertoire racine et la zone des données.

Chapitre 6

Disques au niveau logique : structures DOS de bas niveau

Mots-clefs

Entrée FAT	Une entrée FAT contient une valeur qui indique si le cluster sur lequel elle pointe est libre, réservé, endommagé ou occupé. Si elle indique un secteur occupé, elle pointe en fait le prochain cluster appartenant au fichier qui occupe celui-ci. La dernière entrée de la chaîne ainsi constituée contient une valeur réservée (entre `(F)FF8h` et `(F)FFFh`).
FAT	La FAT est la structure chargée de localiser les fichiers contenus sur le disque.
Lecteur logique	On se voit dans l'obligation de créer des lecteurs logiques lorsqu'on dispose d'un disque dur dont la taille est supérieure au maximum adressable par le DOS. Ces lecteurs logiques font partie de la partition étendue du disque dur et sont donc répertoriés dans la table de partition.
Partition	Une partition de disque dur est l'ensemble des cylindres que le DOS reconnaît comme appartenant au même lecteur logique. Un disque dur comporte au moins une partition, que l'on déclare avec l'utilitaire `FDisk` avant de le formater logiquement.
Partition étendue	Une partition étendue n'est pas bootable mais peut contenir plusieurs lecteurs logiques. Une partition principale doit avoir été créée avant elle.
Secteur boot	Le secteur de boot (*Boot record*) d'un disque ou d'une disquette a deux rôles : il informe le système sur le disque et il contient un programme chargé soit de lancer le système à partir du disque soit d'afficher une chaîne de caractères indiquant que le disque n'est pas bootable.
Secteur de partition	Le secteur de partition d'un disque contient la table de partition, qui informe le système sur les diverses partitions d'un disque et un programme chargé de lire et d'interpréter la table de partition. Ce secteur est caché.
Table de partition	La table de partition contient les adresses de début et de fin de chaque partition du disque, ainsi que sur le type de la partition. Elle se situe dans un secteur caché du disque et est lue par le BIOS avant le secteur de boot. Les disquettes ne contiennent pas de table de partition.

Dans ce chapitre, nous procéderons un peu comme avec la mémoire RAM. Nous étudierons les structures que le DOS met en place pour gérer le formidable espace de travail que représente un disque. Les deux chapitres vont d'ailleurs bien ensemble, puisque nous avons vu que la RAM contenait énormément de données concernant les disques.

Secteur de boot

Sans le secteur de boot (ou une structure lui correspondant), il serait absolument impossible de lire ou d'écrire sur un disque. Le secteur de boot n'est pas seulement destiné à lancer le système du PC, mais contient également des informations vitales pour l'accès au support magnétique. Ce sont en premier lieu ces informations que nous allons étudier.

Données du secteur de boot

Description	*Adresse*
ID constructeur	03h
Octets / secteur	0Bh
Secteurs / cluster	0Dh
Secteurs réservés	0Eh
Nombre de FAT	10h
Nombre d'entrées du répertoire racine	11h
Nombre de secteurs du volume	13h
ID média	15h
Secteurs / FAT	16h
Secteurs / piste	18h
Nombre de têtes	1Ah
Secteurs cachés	1Ch - 20h

Figure 6.1
Informations du secteur de Boot.

Le secteur de boot donne quelques informations intéressantes. L'ID de Média («descripteur de support» en français) permet de connaître immédiatement le type du disque. Les autres données peuvent indiquer si le disque est formaté norma-

lement ou non. On peut également les utiliser en complément les unes des autres pour obtenir certains renseignements.

Selon la documentation officielle publiée par Microsoft, le BIOS Parameter Block se trouverait au complet dans le bloc de paramètres disques. Or, nous avons vu au chapitre 3 (*RAM gérée par le DOS*), qu'il contient entre autres le numéro d'offset disque vers les données, le numéro du premier secteur logique occupé et celui du dernier cluster – tous renseignements qui ne se trouvent pas dans le boot record...

On peut toutefois suppléer à ce manque grâce aux informations qu'il donne. L'offset vers la FAT n'est autre que le nombre de secteurs réservés. L'offset vers le répertoire racine s'obtient en additionnant à l'offset vers la FAT le nombre de secteurs par FAT multiplié par le nombre de FAT. L'offset vers les données ajoute à cette addition le nombre d'entrées fichiers contenues par le répertoire racine multiplié par 32 et divisé par le nombre d'octets par secteur. On peut même afficher le nombre de clusters du disque en divisant le nombre de secteur par le nombre de secteurs par clusters (voir encadré ci-dessous).

```
OfsFAT          Boot[0Eh]
OfsRacine       Boot[10h] * Boot[16h] + OfsFAT
OfsDonnees      (Boot[11h] * 32) / Boot[0Bh] + OfsRacine
NbClust         Boot[13h] / Boot[0Dh]
```

Encadré 6.2
Obtenir des renseignements supplémentaires.

Tout ceci peut vous sembler anecdotique et sans beaucoup d'intérêt. Croyez qu'il n'en est rien. Lorsque l'on travaille à la gestion de disques, ce genre de renseignements est vital : dans ce domaine, l'erreur ne pardonne pas.

Afficher les données du secteur de boot

Le programme `BootRec.Pas` affiche les données du secteur de boot du disque qui se trouve dans le lecteur spécifié en ligne de commande, affiche les données supplémentaires du BIOS Parameter Block et demande si l'on souhaite recommencer l'opération avec un autre lecteur.

Nom de procédure	*Utilité*	*Ligne*
`LitBootRec`	Lit le secteur de boot et initialise le record utilisé	32
`CalculeBiosParam`	Calcule la valeur des données supplémentaires et les affiche	56
`EcritBootRec`	Affiche les données du secteur boot et les données supplémentaires. Demande si l'on veut recommencer avec un autre lecteur	74

Dépendances :

Nom de procédure	Appelle la Proc
Programme	LitBootRec
	EcritBootRec
EcritBootRec	CalculeBiosParam
	LitBootRec
	EcritBootRec

Tableau 6.3
Références croisées de BootRec.Pas.

Listing 6.4
Programme BootRec.Pas. ❶

```
PROGRAM MontreBootRecord; { BootRec.Pas }

USES Crt, Sys;
TYPE Rens = RECORD
             Nom : String[8]; { ID constructeur }
             OpS : Word;      { Octets par Secteur }
             SpC : Byte;      { Secteurs par Cluster }
             Srv : Word;      { Secteurs réservés }
             NbF : Byte;      { Nombre de FAT }
             NbE : Word;      { Entrées ds Racine }
             NbS : Word;      { Nombre de Secteurs }
             ID  : Byte;      { Type Média }
             SpF : Word;      { Secteurs par FAT }
             SpP : Word;      { Secteurs par Piste }
             NbT : Word;      { Nombre de Têtes }
             SC : LongInt;    { Secteurs cachés }
           END;
 BytePtr = ^Byte;

VAR BootRec   : Rens;
    TabOctet  : ARRAY[0..511] OF Byte;
    Lecteur   : String[1];
    NoLect    : Byte;

 {$L Absolute.Obj }
 {$F+}
 PROCEDURE LitSectAbs(Drv : Byte; No, Nb : Word;
                      Tab : BytePtr); EXTERNAL;
 PROCEDURE EcritSectAbs(Drv : Byte; No, Nb : Word;
                        Tab : BytePtr); EXTERNAL;
```

➥

Programme BootRec.Pas (suite). ❷

```
 {$F-}

 PROCEDURE LitBootRecord(Drv : Byte);
 VAR i : Byte;
 BEGIN
   LitSectAbs(Drv, 0, 1, @TabOctet);
   WITH BootRec DO
   BEGIN
     Nom := '';
     FOR i := 0 TO 7 DO
       Nom := Nom + Chr(TabOctet[3 + i]);
     OpS := (TabOctet[$C] SHL 8) + TabOctet[$B];
     SpC := TabOctet[$D];
     Srv := (TabOctet[$F] SHL 8) + TabOctet[$E];
     NbF := TabOctet[$10];
     NbE := (TabOctet[$12] SHL 8) + TabOctet[$11];
     NbS := (TabOctet[$14] SHL 8) + TabOctet[$13];
     ID  := TabOctet[$15];
     SpF := (TabOctet[$17] SHL 8) + TabOctet[$16];
     SpP := (TabOctet[$19] SHL 8) + TabOctet[$18];
     NbT := (TabOctet[$1B] SHL 8) + TabOctet[$1A];
     SC  := (TabOctet[$1F] SHL 24)+
           (TabOctet[$1E] SHL 16)+(TabOctet[$1D] SHL 8)
           + (TabOctet[$1C]);
   END;
END;

PROCEDURE CalculeBiosParam;
VAR OfsFAT, OfsDonnees,
    OfsRacine, NbClust : Word;
BEGIN
  WITH BootRec DO    { Renseignements supplémentaires }
  BEGIN
    OfsFAT := Srv;
    OfsRacine := (NbF * SpF) + OfsFAT;
    OfsDonnees := ((NbE * 32) DIV OpS) + OfsRacine;
    NbClust := (NbS DIV SpC);
  END;
  Window(15, 18, 65, 22); TextAttr := 15 + 4 * 16;
  ClrScr;
  WriteLn(' Offset vers la FAT      : ', OfsFAT:5);
  WriteLn(' Offset vers la Racine   : ', OfsRacine:5);
  WriteLn(' Offset vers les Données : ', OfsDonnees:5);
  Write(' Nombre de Clusters      : ', NbClust:5);
END;
```

➥

Programme BootRec.Pas (suite). ❸

```
PROCEDURE EcritBootRec(Drv : Byte);
BEGIN
  GotoXy(34, 1); TextAttr := 14 + 4 * 16;
  Write(' Boot Record ');
  Window(15, 3, 65, 16); TextAttr := 15 + 4 * 16;
  ClrScr;
  WITH BootRec DO
  BEGIN
    WriteLn;
    WriteLn(' ', Nom, '':5, 'Nom constructeur');
    WriteLn(' ', OpS:5, '':8, 'Octets par Secteur');
    WriteLn(' ', SpC:5, '':8, 'Secteurs par Cluster');
    WriteLn(' ', Srv:5, '':8, 'Secteurs Réservés');
    WriteLn(' ', NbF:5, '':8, 'FAT');
    WriteLn(' ', NbE:5, '':8, 'Entrées dans la Racine');
    WriteLn(' ', NbS:5, '':8, 'Secteurs en tout');
    WriteLn(' ', OctetDecVersHex(ID):5, '':8,
            'ID Média');
    WriteLn(' ', SpF:5, '':8, 'Secteurs par FAT');
    WriteLn(' ', SpP:5, '':8, 'Secteurs par Piste');
    WriteLn(' ', NbT:5, '':8, 'Têtes');
    WriteLn(' ', SC:5, '':8, 'Secteurs sont cachés');
  END;
  CalculeBiosParam;
  ReadLn;
  Window(1, 1, 80, 25);
  GotoXy(5, 24); TextAttr := 15 + 1 * 16;
  Write(' Nouveau Lecteur : ');
  Lecteur[1] := ReadKey; Write(Lecteur[1]); (voir suite)
  IF (Lecteur[1] <> #13) AND (Lecteur[1] <> #27) THEN
  BEGIN
    Lecteur := Upcase(Lecteur[1]);
    NoLect := Ord(Lecteur[1]) - 65;
    Window(1, 1, 80, 25); TextAttr := 15 + 1 * 16;
    ClrScr;
    LitBootRecord(NoLect);
    EcritBootRec(NoLect);                    { Récursive }
  END;
END;

BEGIN
  IF (ParamCount > 0) THEN          { Lecteur en Paramètre }
  BEGIN
    Lecteur := ParamStr(1);
```

➥

Programme BootRec.Pas (suite). ❹

```
    Lecteur[1] :=Upcase(Lecteur[1]);
    NoLect := Ord(Lecteur[1]) - 65;
    Window(1, 1, 80, 25); TextAttr := 15 + 1 * 16;
    ClrScr;
    FillChar(BootRec, SizeOf(BootRec), 0);
    FillChar(TabOctet, SizeOf(TabOctet), 0);
    LitBootRecord(NoLect);
    EcritBootRec(NoLect);
  END
  ELSE
    Write(' Donnez un nom de lecteur en Paramètre.');
END.
```

Programme du secteur de boot

Le secteur de boot contenant en général 512 octets, on se doute bien qu'il comporte autre chose que des données. Il s'agit d'un programme, dont le but varie selon que le disque est ou non bootable. Le programme inscrit sur un disque non bootable affiche quelques octets à l'écran ainsi qu'un message prévenant l'utilisateur qu'il va lui falloir changer de disquette pour relancer le système. Lorsqu'il s'agit d'une disquette système, le programme lit la FAT pour vérifier que les fichiers `IBMDOS.COM` et `IBMBIO.COM` sont présents sur la disquette, charge `IBMBIO.COM` en `07C0h:0000h` et lui passe le contrôle. Les octets `511` et `512` du secteur boot (`01FFh` et `0200h`) ont la valeur `55h` et `AAh` : c'est la signature du secteur.

Lorsqu'un disque n'est pas bootable, il est très simple de comprendre ce que contient le code du secteur de boot à l'aide des informations données plus haut, et d'une connaissance minimum en Assembleur.

```
; Désassemblage du secteur de Boot d'une disquette non
; bootable

Jmp Debut          ; Saut d'initialisation
Nop                ; Rien
Db  'Hal 2001'     ; Données BIOS : ID constructeur
Dw  0200h          ;   Octets/secteur
Db  02h            ;   Secteurs / cluster
Dw  0001h          ;   Secteurs réservés
Db  02h            ;   FAT
Dw  0070h          ;   Entrées dans la racine
Dw  05A0h          ;   Nombre total de secteurs
Db  F9h            ;   ID (disquette 3 1/2, 720 Ko)
Dw  0003h          ;   Secteurs / FAT
```

```
Dw   0009h          ;    Secteurs / piste
Dw   0002h          ;    Nombre de têtes
Dd   00000000h      ;    Secteurs cachés
Debut:              ;    Ici commence le code
Cli                      ; Pas d'INT
Xor Ax,Ax
Mov Ss,Ax                ; Segment de pile à 0
Mov Sp,7bf0h             ; Pointeur de pile
Sti                      ; INT autorisées
Mov Ax,07c0h             ; Segment où transférer le code
Mov Ds,Ax                ; Ds := 07C0h
Mov Si,005bh             ; Index
Nop                      ; Rien
Cld                      ; On incrémente
@1:
LodsB                    ; Charger DS:Si en Al
Or  Al,Al                ; Al = 0 ?
Jz  @2                   ; Oui, c'est fini
Push Si                  ; Sauver Index
Mov Ah,0Eh               ; Ah := 0Eh (fonction)
Mov Bx,0007h             ; Bx := 0007h (Page 0, Blanc)
Int 10h                  ; Ecrit en Noir et blanc le
                           caractère en Al
Pop Si                   ; Récupérer l'index
Jmp @1                   ; Boucler
@2:
Xor Ah,Ah                ; Fonction 00h
Int 16h                  ; Lit un caractère au clavier
Mov Ah,0Fh               ; Fonction 0Fh
Int 10h                  ; Statut vidéo dans Bh
Xor Ah,Ah                ; Fonction 00h
Int 10h                  ; Installer le statut vidéo
Int 19h                  ; Appeler le BootStrap du BIOS
Db 0dh,0ah,0dh,0ah,'Cette disquette n'est pas bootable'
Db 0dh,0ah,'Insérez une disquette DOS et appuyez sur '
Db 'une touche SVP',0dh,0ah
```

Figure 6.5
Le programme du secteur boot sur une disquette non bootable.

Pour connaître le code du programme qui charge le système en mémoire à partir d'une disquette bootable, il suffit d'utiliser DEBUG.

```
C:\>Debug                  ; Charger Débug
- 1 0 0 0 1                ; Charger le Secteur 0 de A:
                           ; en CS:0000
- u 0 1 3                  ; Désassembler le saut
```

```
CS:0000    JMP 0036        ; Adresse du saut : CS:0036
CS:0002    NOP
- u 0036 0176              ; Désassembler de CS:0036 à
                           ; CS:0176
CS:0036    CLI
.....
.....
CS:0176    RET
- d 0177 0200              ; Dumper le reste (messages)
.....
.....
- q                        ; quitter
```

Encadré 6.6
Désassembler le secteur de boot d'une disquette.

Remarque — On peut aussi taper les instructions dans un fichier ASCII et l'envoyer à DEBUG (on peut ainsi travailler directement sur le fichier créé en sortie).

```
C:\> Copy con:Boot.bat
1 0 0 0 1
u 0 1 3
u 0036 0176
d 0177 0200
q
^Z
1 fichier(s) copié(s)
C:\> Debug < Boot.Bat > Boot.Dbg
```

Une fois le désassemblage effectué, on obtient le code suivant :

```
;      Secteur de Boot d'une disquette système      ;
;                (c) Microsoft Corp.                ;

 Jmp   Debut         ; Saut en Début
 Nop                 ; Rien

                     ; Signification            Adresse

 Db    'Hal 2001'    ; ID constructeur          7C03h
 Dw    0200h         ; Octets / Secteur         7C0Bh
 Db    02h           ; Secteurs / Cluster       7C0Dh
 Dw    0001h         ; Secteurs réservés        7C0Eh
 Db    02h           ; Nombre de FAT            7C10h
 Dw    0070h         ; Nombre d'entrées du
                     ; répertoire racine        7C11h
 Dw    02D0h         ; Nombre de secteurs       7C13h
```

```
Db    FDh          ; ID Média                  7C15h
Dw    0002h        ; Secteurs par FAT          7C16h
Dw    0009h        ; Secteurs par piste        7C18h
Dw    0002h        ; Nombre de têtes           7C1Ah
Dd    00000000h    ; Secteurs cachés           7C1Ch
                   ; Espace réservé aux variables du programme
Db    00h, 00h, 00h, 00h, 00h, 00h, 00h, 00h
Db    00h, 00h, 00h, 00h, 00h, 00h, 00h, 12h
Db    00h, 00h, 00h, 00h, 01h, 00h

Debut:

Cli                ; INTs interdites
Xor   Ax, Ax       ; SS := 0000h
Mov   Ss, Ax
Mov   Sp, 7C00h    ; SP := 7C00h
Push  Ss           ; ES := 0000h
Pop   Es
Mov   Bx, 0078h
Lds   Si, Ss:[Bx] ; DS:[SI] := 0000h:[0078h] (= 0522h)
Push  Ds
Push  Si
Push  Bx
Mov   Di, 7C2Bh
Mov   Cx, 000Bh
Cld                ; Incrémentation des index

Charge:
LodSb              ; AL := DS:[SI]
Cmp   Byte Ptr Es:[Di], 00h       ; ES:[DI] = 00h ?
Jz    Stocke                      ; Oui, stocke
Mov   Al, Es:[Di]                 ; Non, AL := ES:[DI]

Stocke:
StoSb                             ; ES:[DI] := AL
Mov   Al, Ah                      ; AL := 00h
Loop  Charge                      ; Boucle 12 fois
Push  Es              ; Ds := 0000h
Pop   Ds
Mov   [Bx+02], Ax                 ; DS:[007Ah] := 0000h
Mov   Word Ptr [Bx], 7C2Bh        ; DS:[0078h] := 7C2Bh
Sti                   ; Ints autorisées
Int   13h             ; réinitialise la disquette
Jb    Erreur2

Mov   Al, [7C10h]         ; AL := Nombre de FATs
CbW                       ; Conversion d'un octet en mot
Mul   Word Ptr [7C16h]    ; Multiplication par le nbre
                          ; de secteurs par FAT
Add   Ax, [7C1Ch]         ; Ax := Ax + NbSectCachés
Add   Ax, [7C0Eh]         ; Ax := Ax + NbSectRsrvés
Mov   [7C3Fh], Ax         ; DS:[7C3Fh] = Secteurs réservés
```

```
Mov    [7C37h], Ax       ; DS:[7C37h] = DS:[7C3Fh]

Mov    Ax, 0020h         ; Ax := 32
Mul    Word Ptr [7C11h]  ; Ax := Ax * NbEntreesRacine
Mov    Bx, [7C0Bh]       ; Bx := OctetsParSecteur
Add    Ax, Bx            ; Ax := Ax + Bx
Dec    Ax
Div    Bx                ; Ax := Ax / 512
Add    [7C37h], Ax       ; DS:[7C37h] = Secteurs réservés
                         ; plus secteurs de la Racine
Mov    Bx, 0500h         ; Buffer en 0000h:0500h
Mov    Ax, [7C3Fh]       ; Ax := Secteurs réservés
Call   ConversionSecteurs
Mov    Ax, 0201h
Call   LitSecteur
Jb     Erreur1

Mov    Di, Bx      ; Comparaison des noms de fichiers
                   ; système avec les deux premières
                   ; entrées de la racine
Mov    Cx, 000Bh
Mov    Si, 7DDBh         ; 7DDBh := Ofs('IO.SYS')
RepZ
CmpSb
JnZ    Erreur1
Lea    Di, [Bx+20h]      ; Deuxième entrée fichier en
                         ; 0000:0520h
Mov    Si, 7DE6h         ; 7DE6h := Ofs('MSDOS.SYS')
Mov    Cx, 000Bh
RepZ
CmpSb
Jz     Ok

Erreur1:          ; Gestion des erreurs
Mov    Si, 7D77h         ; 7D77h := Ofs(ErrMsg1)

Erreur:
Call   AfficheMsg
Xor    Ah, Ah            ; Lecture clavier
Int    16h
Pop    Si                ; Reset du PC
Pop    Ds
Pop    [Si]
Pop    [Si+02]
Int    19h

Erreur2:
Mov    Si, 7DC7h         ; 7DC7h := Ofs(ErrMsg2)
Jmp    Erreur

Ok:
Mov    Ax, [051Ch]       ; Ax := Taille de IO.SYS
```

```
Xor    Dx, Dx
Div    Word Ptr [7C0Bh]   ; Ax := Ax / Octets par Secteur
Inc    Al                 ; Ax := Ax + 1
Mov    [7C3Ch], Al        ; Nombre de secteurs d'IO.SYS en
                          ; 0000:7C3Ch
Mov    Ax, [7C37h]        ; Ax := Secteurs réservés +
                          ; secteurs de la racine
Mov    [7C3Dh], Ax        ; Ax en 0000:7C3Dh
Mov    Bx, 0700h          ; Offset buffer

Lit_IO_SYS:
Mov    Ax, [7C37h]        ; No de secteur
Call   ConversionSecteurs
Mov    Ax, [7C18h]        ; Ax := Secteurs par piste
Sub    Al, [7C3Bh]        ; Ax := Ax - ?
Inc    Ax
Cmp    [7C3Ch], Al        ; AL = Nombre de secteurs
                          ; d'IO.SYS ?
JnB    Oui
Mov    Al, [7C3Ch]

Oui:
Push   Ax
Call   LitSecteur
Pop    Ax
Jb     Erreur2
Sub    [7C3Ch], Al        ; Décrémente Nbre de secteurs
                          ; du fichier
Jz     DonneLaMain
Add    [7C37h], Ax        ; Incrémente numéro de secteur
Mul    Word Ptr [7C0Bh]   ; Ax := Ax * Octets par secteur
Add    Bx, Ax             ; Incrémente l'adresse du buffer
Jmp    Lit_IO_SYS

DonneLaMain:
Mov    Ch, [7C15h]        ; CH := ID Média
Mov    Dl, [7DFDh]        ; DL := Seg('IO.SYS')
Mov    Bx, [7C3Dh]        ; BX := Secteurs réservés +
                          ; secteurs de la racine
Jmp    0070h:0000h        ; Exécute IO.SYS

AfficheMsg      Proc     Near

DebutAffichage:
LodSb                     ; Charge DS:SI en AL
Or     Al, Al             ; Si Al = 0, Fin de l'affichage
Jz     FinProc
Mov    Ah, 0Eh            ; Affichage d'un caractère
Mov    Bx, 0007h          ; page 00, en blanc sur noir
Int    10h
Jmp    DebutAffichage
```

```
Conversion Secteurs:        ; Convertir un N° de secteur
                            ; logique en Face, Cylindre,
                            ; et Secteur
Xor    Dx, Dx
Div    Word Ptr [7C18h]     ; Ax := Ax / Secteurs par piste
Inc    Dl                   ; Incrémenter le reste (en DX)
Mov    [7C3Bh], Dl          ; 0000:7C3Bh := Reste
Xor    Dx, Dx
Div    Word Ptr [7C1Ah]     ; Ax := Ax / Nombre de têtes
Mov    [7C2Ah], Dl          ; Reste en 0000:7C2Ah
Mov    [7C39h], Ax          ; Résultat en 0000:7C39h

FinProc:
Ret
AfficheMsg     EndP

LitSecteur     Proc    Near
Mov    Ah, 02h              ; Lire un secteur
Mov    Dx, [7C39h]
Mov    Cl, 06h
Shl    Dh, Cl
Or     Dh, [7C3Bh]
Mov    Cx, Dx
XChg   Ch, Cl               ; Ch := Cylindre, Cl := Secteur
Mov    Dl, [7DFDh]          ; Dl := No du Drive
                            ; (généralement 0 ou 80h)
Mov    Dh, [7C2Ah]          ; Dh := No de tête
Int    13h                  ; Lecture
Ret
LitSecteur     EndP

; Message d'erreur numéro 1          Offset = 7D77h
Db 0dh,0ah,'Disque sans système ou erreur disque ',
Db 0dh,0ah,'Remplacez et appuyez sur une touche ',
Db 0dh,0ah,00h

; Message d'erreur numéro 2          Offset = 7DC7h
Db     0dh,0ah,'Erreur d'amorçage',0dh,0ah,00h

Db     'IO      SYS'    ;            Offset = 7DBBh
Db     'MSDOS   SYS'    ;            Offset = 7DE6h
Db     00h, 00h, 00h, 00h, 00h, 00h, 00h
Db     00h, 00h, 00h, 00h, 00h, 00h
Db     55h, AAh         ;            Signature
```

Les quelques programmes de formatage qui existent sur le marché (par exemple `Safe Format` des *Norton Utilities* et `PcFormat` de *PCTOOLS*) utilisent tous la même méthode d'écriture du secteur zéro du disque à formater. S'il leur faut formater un disque système, ils demandent à l'utilisateur d'introduire une disquette bootable en `A:`, dont ils recopient le secteur zéro et les fichiers `IBMDOS.COM` et `IBMBIO.COM` sur la disquette qu'ils sont chargés de formater. Inversement, c'est leur

propre code qu'ils copient sur une disquette non système. Le programme qui suit s'inspire de ces utilitaires : il lit le secteur zéro d'une disquette (système ou non), met les données à jour en fonction de la disquette sur laquelle le recopier et effectue la copie. Ce n'est pas autre chose, en fin de compte, qu'un élément d'un programme de formatage.

Type	*Nom Proc*	*Description*	*Ligne*
P	`FormatDsktte`	Complète les données du Boot Record que le BIOS ne fournit pas	*37*
F	`DonneesDrv`	Utilise l'Int `13h`, fonction `08h` pour obtenir des renseignements du drive	*75*
P	`LitBootRec`	Lit le secteur boot et en enlève les données	*101*
P	`MetAJourDonnees`	Mise à jour des données du nouveau secteur boot	*109*
P	`EcritBootRec`	Ecrit le nouveau secteur boot sur la disquette	*133*
	`Programme`		*140*

Dépendances :

Procédure	*Appelle*
`Programme`	`FormatDsktte`
	`DonneesDrv`
	`LitBootRec`
	`MetAJourDonnees`
	`EcritBootRec`
`LitBootRec`	`LitSectAbs`
`EcritBootRec`	`EcritSectAbs`

Tableau 6.7
Références croisées de BootCopy.Pas.

Listing 6.8
Programme BootCopy.Pas. ❶

```
PROGRAM CopieBootRecord;           { BootCopy.Pas }

USES Dos, Crt;

TYPE
 Donnees   = RECORD
             Nom    : String[8];
             OpS    : Word;
```

➥

Programme BootCopy.Pas (suite). ❷

```
                SpC    : Byte;
                Srv    : Word;
                NbF    : Byte;
                NbE,
                NbS    : Word;
                ID     : Byte;
                SpF,
                SpP,
                NbT    : Word;
                SC     : LongInt;
              END;            { Record }
  BytePtr   = ^Byte;

 VAR
  BootRec         : Donnees;
  TabOctet        : ARRAY[0..511] OF Byte;
  Lecteur         : String[1];
  NoLect,
  Erreur          : Byte;

  {$L Absolute.Obj}
  {$F+}
  PROCEDURE LitSectAbs(Drv : Byte; No, Nb : Word;
                       Tab : BytePtr); EXTERNAL;
  PROCEDURE EcritSectAbs(Drv : Byte; No, Nb : Word;
                         Tab : BytePtr); EXTERNAL;
  {$F-}

  PROCEDURE FormatDsktte(Drv : Byte);
  BEGIN
    IF (ParamStr(2) = '720') THEN
    BEGIN
      BootRec.NbS := 1440;
      BootRec.ID := $F9;
      BootRec.SpF := 3;
      BootRec.SpP := 9;
      BootRec.NbE := 112;
      BootRec.SpC := 2;
    END
    ELSE IF (ParamStr(2) = '360') THEN
    BEGIN
      BootRec.NbS := 720;
      BootRec.ID := $FD;
      BootRec.SpF := 2;
      BootRec.SpP := 9;
```

➥

Programme BootCopy.Pas (suite). ❸

```
      BootRec.NbE := 112;
      BootRec.SpC := 2;
    END
    ELSE IF (ParamStr(2) = '144') THEN
    BEGIN
      BootRec.NbS := 2880;
      BootRec.ID := $F0;
      BootRec.SpF := 9;
      BootRec.NbE := 224;
      BootRec.SpC := 1;
    END
    ELSE IF (ParamStr(2) = '120') THEN
    BEGIN
      BootRec.NbS := 2400;
      BootRec.ID := $F9;
      BootRec.SpF := 7;
      BootRec.NbE := 224;
      BootRec.SpC := 1;
    END;
  END;
  FUNCTION DonneesDrv(Drv : Byte) : Byte;
  VAR Regs  : Registers;
  BEGIN
    Regs.Dl := Drv;
    Regs.Ah := $08;
    Intr($13, Regs);
    IF (Regs.Flags AND 1 = 1) THEN
      DonneesDrv := $FF
    ELSE
    BEGIN
      Write(Regs.Bl);
      WITH BootRec DO
      BEGIN
        NbT := Regs.Dh;
        NbS := (Regs.Ch * Regs.Cl) * (NbT + 1);
        SpP := Regs.Cl;
        OpS := 128 * (Mem[Regs.Es:Regs.Di + 3] SHL 1);
        SC  := 0;
        NbF := 2;
        Srv := 1;
        Nom := 'GB&PSI 1';
      END;
      DonneesDrv := Regs.Dl
    END;
  END;
```

➥

Programme BootCopy.Pas (suite). ❹

```
PROCEDURE LitBootRec(Drv : Byte);
VAR i  : Byte;
BEGIN
  LitSectAbs(Drv, 0, 1, @TabOctet);
  FOR i := 3 TO $1F DO
    TabOctet[i] := 0;
END;

PROCEDURE MetAJourDonnees;
VAR i  : Byte;
BEGIN
  WITH BootRec DO
  BEGIN
    FOR i := 3 TO $A DO
      TabOctet[i] := Ord(Nom[i - 2]);
    TabOctet[$B]  := Lo(OpS); TabOctet[$C] := Hi(OpS);
    TabOctet[$D]  := SpC;
    TabOctet[$E]  := Lo(Srv); TabOctet[$F] := Hi(Srv);
    TabOctet[$10] := NbF;
    TabOctet[$11] := Lo(NbE); TabOctet[$12] := Hi(NbE);
    TabOctet[$13] := Lo(NbS); TabOctet[$14] := Hi(NbS);
    TabOctet[$15] := ID;
    TabOctet[$16] := Lo(SpF); TabOctet[$17] := Hi(SpF);
    TabOctet[$18] := Lo(SpP); TabOctet[$19] := Hi(SpP);
    TabOctet[$1A] := Lo(NbT); TabOctet[$1B] := Hi(NbT);
    TabOctet[$1C] := (SC SHR 24);
    TabOctet[$1D] := (SC SHR 16);
    TabOctet[$1E] := (SC SHR 8);
    TabOctet[$1F] := SC OR
                     ((SC SHR 24) AND (SC SHR 16) AND
                      (SC SHR 8));
  END;
END;

PROCEDURE EcritBootRec(Drv : Byte);
BEGIN
  Write('Mise à jour du Boot Record...');
  EcritSectAbs(Drv, 0, 1, @TabOctet);
  Write(' Ok');
END;

BEGIN
 IF (ParamCount < 2) THEN
```

➥

Programme BootCopy.Pas (suite). ❺

```
  BEGIN
    WriteLn(' Ligne de commande nécessaire : ');
            720(144/360/120) ');
    WriteLn('':10, ' Lecteur = A:, B:, etc.');
    WriteLn('':10, ' 720 = 3p 1/2, 720 Ko,
            120 = 5p.1/4, 1.2 Mo, etc.');
  END
  ELSE
  BEGIN
    Lecteur := ParamStr(1);
    Lecteur[1] := Upcase(Lecteur[1]);
    NoLect := Ord(Lecteur[1]) - 65;
    WriteLn(' Insérez une disquette DOS en A:');
    WriteLn(' Puis, appuyez sur <Ÿ ');
    ReadLn;
    LitBootRec(0);
    Write(' Insérez la disquette où recopier le
          Boot Record en ');
    WriteLn(Lecteur[1], ':  puis appuyez sur <Ÿ');
    ReadLn;
    Erreur := DonneesDrv(NoLect);
    IF (Erreur <> $FF) THEN
    BEGIN
      FormatDsktte(NoLect);
      MetAJourDonnees;
      EcritBootRec(NoLect);
      WriteLn(' Opération terminée');
    END
    ELSE
    BEGIN
      Write(' Erreur : Abandon...');
      Halt(1);
    END;
  END;
END.
```

FAT (File Allocation Table)

La FAT est probablement la structure qui fédère et qui divise le plus les systèmes d'exploitation. Elle les fusionne par l'absolue nécessité de disposer d'une table des

matières des fichiers. Elle les désunit par la diversité des moyens envisagés pour résoudre le problème du rangement des fichiers sur un support magnétique.

Utilité de la FAT

La FAT n'est autre qu'une table des matières, ou plutôt – on serait plus proche de la vérité – un index. Lorsque l'on achète un livre d'informatique, il est rare qu'on le lise d'une seule traite. Les lecteurs s'en servent généralement plus comme d'une source de renseignements, ce qu'on appelle un manuel de référence. Dans un tel cas, on se reporte à l'index et l'on va regarder, par exemple, à l'entrée BIOS, où l'on voit :

```
BIOS - Interruptions
BIOS - Interruptions - Disquettes
BIOS - Interruptions - Disquettes - Fonction 00h
...
```

La FAT fonctionne à peu près de la même manière. Elle n'indique pas seulement à quel endroit du disque se trouve un fichier, mais à quel endroit se trouve la première partie du fichier, la seconde, etc. On peut rapprocher cela d'une bibliothèque où le fichier des œuvres indiquerait le rayon dans lequel est rangé le premier chapitre d'un livre, puis un autre rayon pour le second chapitre, et ainsi de suite jusqu'à la conclusion. Pour résumer cela, on dit que la FAT est la structure chargée de localiser les fichiers contenus sur le disque.

Malgré cette organisation farfelue (mais difficilement améliorable), l'utilité de la FAT est réelle. Sans elle ou un système approchant (comme celui d'UNIX, par ailleurs bien plus performant), le disque serait, là encore, inutilisable.

Fonctionnement de la FAT

Nous venons de brosser à grands traits les principes de fonctionnement de la FAT. Allons un peu plus loin dans la théorie avant d'en aborder les caractéristiques exactes.

La FAT peut être vue sous l'aspect d'un tableau de pointeurs. Ce n'est pas la première structure DOS à utiliser cette structure de données, et moins encore la dernière. Chaque entrée de la FAT possède un numéro d'ordre. Ces entrées contiennent chacune un numéro identifiant la prochaine entrée. Jusqu'ici, cela a l'air simple. Mais il faut raffiner. En effet, ces numéros sont à la fois un index et un numéro de cluster : l'entrée n° 2 de la FAT représente donc le cluster n° 2 du disque et le second enregistrement du tableau. Si cette entrée contient un 5, cela signifie qu'il faut aller à l'entrée n° 5 de la FAT pour connaître le troisième cluster où se trouve le fichier, mais cela veut également dire que le second cluster occupé par le fichier est le n° 5... Et le premier cluster du fichier, comment l'identifie-t-on ?

Ce n'est pas la FAT qui s'en charge, mais un enregistrement de l'entrée-fichier (voir le chapitre suivant). Pour l'instant, il faut surtout retenir qu'il y a une entrée FAT par cluster disponible sur le disque. Or, chaque cluster contient entre 1 et 8 secteurs selon les formats de disque (voir le chapitre 5, *Disques au niveau logique : le plan d'un disque*). Et, naturellement, un fichier peut ne pas occuper *réellement* un cluster complet...

Voyons maintenant le côté technique de tout ceci. Pour commencer, nous allons examiner le fonctionnement de la FAT sur un disque dur de plus de 10 Mo. Ces disques ont en effet l'avantage d'être divisés en clusters de 4 secteurs et de posséder une FAT au format 16 bits.

FAT au format 16 bits

Les deux premières entrées de la FAT sont réservées : le premier octet de la première entrée contient l'ID de média, tandis que le second, comme les deux octets de la deuxième entrée, contient la valeur `FFh`.

Signification	*Valeur correspondante*
Cluster libre	`0000h`
Cluster endommagé	`FFF7h`
Dernier cluster d'un fichier	`FFF8h-FFFFh`
Réservé	`FFF0h-FFF6h`
Sans signification (inutilisé)	`0001h`
Prochain cluster du fichier	Toute autre valeur

Tableau 6.9
Valeurs réservées de la FAT 16 bits.

La FAT commence donc réellement à l'entrée n° 2 (les entrées sont numérotées à partir de 0). Chaque mot d'une entrée FAT indique l'index auquel on trouvera la valeur du prochain cluster occupé par le fichier en cours. Lorsqu'une entrée est libre, elle contient la valeur `0000h`. Si le cluster qu'elle désigne est endommagé, la valeur réservée est de `FFF7h`. S'il s'agit du dernier cluster d'un fichier, toute valeur comprise entre `FFF8h` et `FFFFh` peut être inscrite dans l'entrée en question. Enfin, les valeurs `FFF0h` à `FFF6h` sont réservées et la valeur `0001h` n'est jamais employée (tout simplement parce que les clusters sont numérotés à partir de 2).

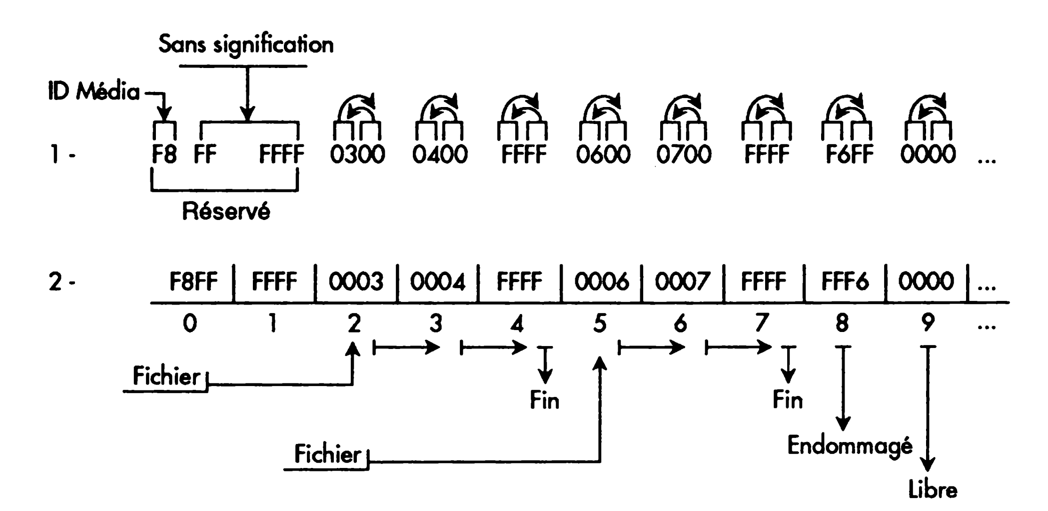


Figure 6.10
La FAT d'un disque dur (format 16 bits).

Le format 16 bits permet un accès relativement simple aux entrées FAT : il suffit en fait de multiplier le numéro du premier cluster par deux pour obtenir l'index de l'entrée où aller lire le prochain numéro de cluster. Toutefois, comme on accède à la FAT en lisant un secteur absolu du disque, il faut avant tout connaître le numéro de secteur à lire : pour cela, on divise le numéro de cluster connu par la moitié du nombre d'octets par secteur (la FAT contient des mots et non des octets), et l'on ajoute un au résultat (l'offset vers la FAT est toujours de un). Lorsque l'on cherchera l'entrée FAT, on calculera également le déplacement de l'index à l'intérieur du secteur par la fonction `Modulo` qu'offre le Pascal aussi bien que l'Assembleur (voir *figure 1.4*).

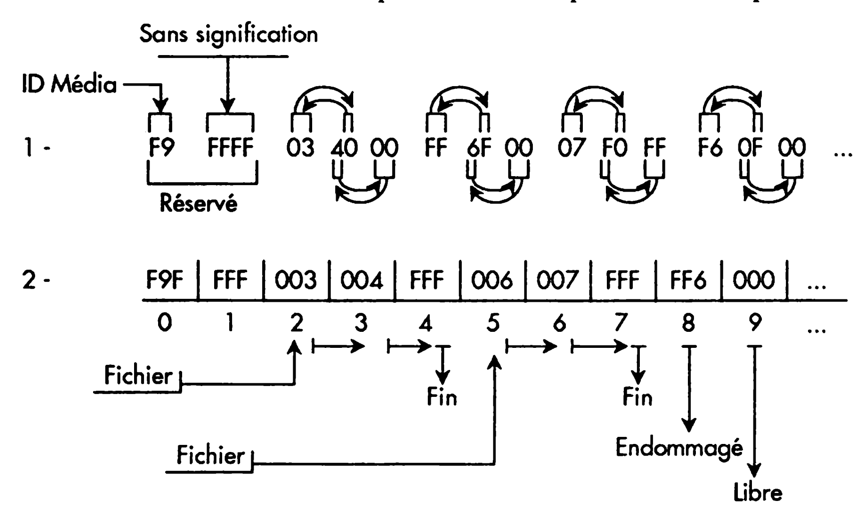
```
NoSectFAT    := ( NoClust DIV (OpS SHR 1)) + 1;
NoIndexFAT   := ( NoClust MOD (OpS SHR 1)) SHL 1;
NoClustSuivt := (FAT[NoIndexFAT+1] SHL 8) +
                (FAT[NoIndexFAT]);
```

Figure 6.11
Calcul du numéro d'index d'une entrée FAT 16 bits.

Remarque — `NoClust` est le premier numéro de cluster d'un fichier. `OpS` représente le nombre d'octets par secteur. `NoSectFAT` est le numéro de secteur logique où se trouve l'entrée FAT correspondant au premier numéro de cluster d'un fichier. NoIndexFAT est l'index relatif au secteur où l'on trouvera l'entrée FAT recherchée. `NoClustSuivt` est le numéro de cluster contenu dans l'entrée FAT, qui indique également la prochaine entrée FAT.

FAT au format 12 bits

Le format 12 bits est employé par le DOS pour les disquettes et les disques durs de moins de `10 Mo`. Il est nettement plus complexe à comprendre et à gérer que le format 16 bits. En revanche, il prend moins de place sur le disque.



1 : la FAT 12 bits telle qu'elle est enregistrée, et les manipulations nécessaires à son interprétation
2 : la FAT interprétée

Figure 6.12
La FAT d'une disquette (format 12 bits).

La particularité principale de ce format est que chaque entrée de la FAT fait référence à deux clusters et donc à deux autres entrées. En effet, le micro-ordinateur gère des nombres de 8, 16 ou 32 bits. Une entrée *physique* d'une FAT au format 12 bits fait donc malgré tout 16 bits. Parmi ces 16 bits, il faut en retirer 4 pour obtenir un numéro de cluster correct. Ces 4 bits sont liés à l'entrée suivante ou à la précédente. On peut donc dire qu'une entrée (16 bits) contient la valeur d'un cluster (12 bits) et demi (4 bits). Mais il ne s'agit jamais du même demi-cluster...

Signification	*Valeur correspondante*
Cluster libre	000h
Cluster endommagé	FF7h
Dernier cluster d'un fichier	FF8h-FFFh
Réservé	FF0h-FF6h
Sans signification (inutilisé)	001h
Prochain cluster du fichier	Toute autre valeur

Tableau 6.13
Valeurs réservées de la FAT 12 bits.

Plusieurs nouveaux éléments entrent en jeu pour le calcul de l'index auquel on trouvera la valeur du nouveau cluster. Il faut avant tout se représenter la FAT comme un tableau d'octets commençant à l'indice 0.

Le cluster connu fournit le numéro d'index. Le même cluster est ensuite divisé par deux pour obtenir le numéro de l'octet à lire dans l'index. On ajoute à cette nouvelle valeur la valeur d'index et l'on obtient alors la valeur absolue de l'indice auquel lire le tableau. C'est alors que l'on calcule le numéro de secteur logique auquel appartient l'index en le divisant par le nombre d'octets par secteur, et en ajoutant un au résultat. On calcule ensuite le déplacement dans le secteur auquel correspond l'index en récupérant le reste (modulo) de la division de l'index par le nombre d'octets par secteur.

Comme les valeurs entreposées dans ce tableau sont des mots et non des octets, on lit un mot à l'indice+1 que l'on décale de huit bits vers la gauche avant d'y ajouter la valeur lue à l'indice. Ensuite, si la division du cluster par deux ne tombe pas juste (si le cluster est impair), on décale le mot obtenu de 4 bits vers la droite. Si au contraire il est pair, on fait un AND entre ce mot et la valeur 0FFFh, ce qui permet de récupérer les douze bits de poids faible. Tout ceci paraît confus à première lecture : il n'est en effet pas simple d'expliquer ce mécanisme.

```
OfsFAT    := OfsFAT + Cluster
DemiOfs   := Cluster SHR 1
OfsFAT    := OfsFAT + DemiOfs
NoSectFAT:= (OfsFAT DIV OpS)+1     { Secteur FAT à lire }
OfsFAT    := OfsFAT MOD OpS        { Indice définitif }
MotLu     := (FAT[OfsFAT+1] SHL 8) + FAT[OfsFAT]
IF (Cluster MOD 2 = 0) THEN        { Entrée paire }
  MotLu := MotLu And $0FFF
ELSE                               { Entrée impaire }
  MotLu := MotLu SHR 4;
```

Figure 6.14
Calcul de l'index d'une entrée-FAT 12 bits.

Remarque — `OfsFAT` est habituellement 0 (au début). `Cluster` est le numéro de cluster connu. `DemiOfs`, ajouté à `OfsFAT`, permet d'obtenir l'offset réel (en valeur absolue, c'est-à-dire sans tenir compte de la division en secteurs) où lire la nouvelle valeur. `NoSectFAT` identifie le numéro de secteur logique à lire. `MotLu` est le mot sur 16 bits contenu dans l'entrée FAT. On le traite ensuite de telle manière que l'on obtienne une valeur sur 12 bits, qui est le nouveau numéro de cluster.

Lire les valeurs de la FAT

Lorsque l'on souhaite écrire un programme utilisant la FAT, on n'a que l'embarras du choix : cela va de la vérification de l'identité des deux FAT au "mappage" disque en passant par les programmes de diagnostics divers. Sans compter qu'il est toujours possible d'en afficher les valeurs. Mais illustrer le fonctionnement de la FAT en 300 lignes de programme est relativement complexe. Cela tient essentiellement à l'étrangeté des principes auxquels elle obéit, et dont nous n'aurons fait le tour complet qu'après avoir vu les entrées fichiers. Pour le moment, il suffit de savoir que tout change selon que l'on cherche un fichier ou un répertoire.

Le programme `FAT.Pas` a pour objet d'afficher les numéros de cluster dans lesquels se trouve un fichier. Cela impose de connaître le chemin d'accès absolu du fichier : le programme doit en effet lire les numéros de cluster des répertoires avant ceux du fichier. Le chemin est donc décomposé en noms de répertoire et nom de fichier. `FAT.Pas` cherche alors les numéros des clusters où se trouvent les répertoires, puis ceux qui appartiennent au fichier.

Nom	*Type*	*Description*	*Ligne*
`LitSectBoot`	P	Lit le secteur de boot et initialise des variables	*36*
`CheminAChercher`	F	Décompose la ligne de commande en noms de répertoire et de fichier	*55*
`ChercheEntreeFic`	F	Cherche l'entrée fichier dont le nom est précisé dans un cluster de répertoire	*72*
`Egalite`	F	Compare deux noms de fichiers	*79*
`SupprimeBlancs`	F	Mise en forme des noms de fichiers à comparer	*81*
`ChercheEntreeFAT`	F	Renvoie la valeur d'une entrée FAT	*158*
`ChercheRepertoire`	F	Cherche une entrée fichier de répertoire et son numéro de cluster	*188*

➡

(suite du tableau)

Nom	Type	Description	Ligne
ChercheFichier	F	Cherche une entrée fichier et ses numéros de cluster	241
Ecran	P	Dessine l'écran	267
Programme			280

Dépendances :

Procédure	Appelle procédure
Programme	LitSectBoot
	CheminAChercher
	ChercherEntreeFAT
	ChercheRepertoire
	ChercheFichier
	Ecran
LitSectBoot	LitSectAbs
ChercherEntreeFAT	LitSectAbs
ChercheRepertoire	ChercheEntreeFic
	ChercheEntreeFAT
ChercheEntreeFic	LitSectAbs
	\Egalite
	\Egalite \...\SupprimeBlancs
ChercheFichier	ChercheEntreeFic
	ChercheEntreeFAT

Tableau 6.15
Références croisées de FAT.Pas.

Listing 6.16
Programme FAT.Pas. ❶

```
PROGRAM TrouveNosClust;              { FAT.Pas }

USES Dos, Crt;

TYPE Tableau    = ARRAY[0..511] OF Byte;
     BytePtr    = ^Byte;
```

➥

Programme FAT.Pas (suite). ❷

```
     EntreeFic = RECORD
                  Nom     : String[12];
                  Attr    : Byte;
                  Taille  : LongInt;
                  Cluster : Word;
                 END;
VAR
 TabFic, TabFat      : Tableau;
 Entree              : EntreeFic;
 Drv, i, Sauve       : Byte;
 PrecDir, Clust, OpS,
 Fin, SpC, OfsDonnees,
 OfsRacine, NoSect   : Word;
 SeizeBits, Trouve   : BOOLEAN;
 Chaine, Morceau,
 Fichier             : String;
 Attribut            : String[48];
 D                   : DirStr;
 N                   : NameStr;
 E                   : ExtStr;
 {$L Absolute.Obj}
 {$F+}
 PROCEDURE LitSectAbs(Drv : Byte; No, Nb : Word;
                      Tab : BytePtr); EXTERNAL;
 PROCEDURE EcritSectAbs(Drv : Byte; No, Nb : Word;
                        Tab : BytePtr); EXTERNAL;
 {$F-}

 PROCEDURE LitSectBoot(Drv : Byte);
 VAR Srv, SC, SpF, NbF, NbS, NbE : Word;
 BEGIN
   LitSectAbs(Drv, 0, 1, @TabFic);
   OpS := (TabFic[$C] SHL 8) + TabFic[$B];
   SpC := TabFic[$D];
   Srv := (TabFic[$F] SHL 8) + TabFic[$E];
   NbF := TabFic[$10];
   NbE := (TabFic[$12] SHL 8) + TabFic[$11];
   NbS := (TabFic[$14] SHL 8) + TabFic[$13];
   SpF := (TabFic[$17] SHL 8) + TabFic[$16];
   SC  := (TabFic[$1F] SHL 24) + (TabFic[$1E] SHL 16) +
          (TabFic[$1D] SHL 8) + TabFic[$1C];
   SeizeBits := ((NbS DIV SpC) > 4078);
   OfsRacine := ((NbF * SpF) + Srv);
   OfsDonnees := OfsRacine + ((NbE * 32) DIV OpS);
   FillChar(TabFic, SizeOf(TabFic), 0);
 END;
```

➡

Programme FAT.Pas (suite). ❸

```
FUNCTION CheminAChercher(VAR Chemin : String) : String;
VAR Chaine : String;
BEGIN
  IF Pos('\', Chemin) <> 0 THEN
  BEGIN
    Delete(Chemin, 1, Pos('\', Chemin));
    IF Pos('\', Chemin) <> 0 THEN
      Chaine := Copy(Chemin, 1, Pos('\', Chemin) - 1)
    ELSE
      Chaine := Chemin;
  END
  ELSE
    Chaine := '';
  CheminAChercher := Chaine;
END;

FUNCTION ChercheEntreeFic(Drv : Byte; Sect : Word;
                          VAR Nom : String) : Word;
VAR i           : Byte;
    j           : Word;
    Trouve      : Boolean;
    NomCherche,
    Fichier     : String;

  FUNCTION Egalite(S1, S2 : String) : Boolean;

    FUNCTION SupprimeBlancs(S : String) : String;
    VAR i  : Byte;
    BEGIN
      i := Length(S);
      REPEAT
        IF (S[i] = #32) THEN
          Delete(S, i, 1);
        Dec(i);
      UNTIL (S[i] <> #32) OR (i = 1);
      IF ((Pos('.', S) <> 9) AND (Pos('.', S) > 2)) THEN
      BEGIN
        i := Pos('.', S);
        REPEAT
          Insert(' ', S, i);
          Inc(i);
        UNTIL (i = 9);
      END;
      SupprimeBlancs := S;
    END;
```

➥

Programme FAT.Pas (suite). ❹

```
  VAR i   : Byte;
     Ok   : Boolean;
  BEGIN              { Egalite }
    Ok := True;
    S1 := SupprimeBlancs(S1);
    S2 := SupprimeBlancs(S2);
    IF (Length(S1) = Length(S2)) THEN
    BEGIN
      i := Length(S1);
      WHILE ((Ok) AND (i > 0)) DO
      BEGIN
        IF (Upcase(S1[i]) <> Upcase(S2[i])) THEN
          Ok := False;
        Dec(i);
      END;
    END
    ELSE
      Ok := False;
    Egalite := Ok;
  END;

BEGIN              { ChercheEntreeFic }
  i := 1; j := 0; Trouve := False; Fichier := Nom;
  LitSectAbs(Drv, Sect, 1, @TabFic);
  REPEAT
    NomCherche := '';
    WHILE (Length(NomCherche) < 11) DO
      NomCherche := NomCherche +
      Chr(TabFic[j + Length(NomCherche)]);
    IF (Pos('.', Nom) <> 0) THEN
      NomCherche := Copy(NomCherche, 1, 8) + '.' +
                    Copy(NomCherche, 9, 12);
    IF (Egalite(NomCherche, Nom)) THEN
      Trouve := True
    ELSE
      Inc(j, 32);
    IF (j > (512 - 32)) THEN
    BEGIN
      Inc(i);
      j := 0;
      Inc(Sect);
      LitSectAbs(Drv, Sect, 1, @TabFic);
    END;
  UNTIL ((Trouve) OR (i > SpC) OR (NomCherche[1] = #0));
  IF Trouve THEN
```

➥

Programme FAT.Pas (suite). ❺

```
    BEGIN
    WITH Entree DO
    BEGIN
      Nom     := NomCherche;
      Attr    := TabFic[j + $B];
      Cluster := (TabFic[j+$1B] SHL 8)+TabFic[j+$1A];
      Taille  := (TabFic[j+$1F] SHL 24)+
                 (TabFic[j+$1E] SHL 16)+
                 (TabFic[j+$1D] SHL 8) +
                 (TabFic[j+$1C]);
    END;
    ChercheEntreeFic := Entree.Cluster;
  END
  ELSE
    ChercheEntreeFic := 0;
END;

FUNCTION ChercheEntreeFAT(Drv : Byte;
                          VAR Clust : Word) : Word;
VAR NoSect, NoIndex,
    OfsFAT, DemiOfs,
    Resultat       : Word;
BEGIN
  Resultat := 0;
  IF SeizeBits THEN
  BEGIN
    NoSect := (Clust DIV (OpS SHR 1)) + 1;
    LitSectAbs(Drv, NoSect, 1, @TabFat);
    NoIndex := (Clust MOD (OpS SHR 1)) SHL 1;
    Resultat := (TabFat[NoIndex + 1] SHL 8) +
                TabFat[NoIndex];
  END
  ELSE
  BEGIN
    OfsFAT := Clust;
    DemiOfs := (Clust SHR 1);
    OfsFAT := OfsFAT + DemiOfs;
    NoSect := (OfsFAT DIV OpS) + 1;
    LitSectAbs(Drv, NoSect, 1, @TabFat);
    OfsFAT := OfsFAT MOD OpS;
    Resultat := (TabFat[OfsFAT+1] SHL 8)+
                TabFat[OfsFAT];
    IF (Clust MOD 2 = 0) THEN
      Resultat := Resultat AND $0FFF
```

➡

Programme FAT.Pas (suite). ❻

```
    ELSE
      Resultat := Resultat SHR 4;
  END;
  ChercheEntreeFAT := Resultat;
END;

FUNCTION ChercheRepertoire(Drv : Byte; Clust : Word;
                           Nom : String) : Word;
VAR NoSect, i, Premier, Cluster : Word;
    Lig                         : Byte;
    Continuer, Trouve           : Boolean;
BEGIN
  Trouve := False; i := 0; Continuer := True;
  IF (Clust < 2) THEN
  REPEAT
    Cluster := ChercheEntreeFic(Drv,OfsRacine+i,Nom);
    IF (Cluster > 0) THEN
      Trouve := True
    ELSE
      Inc(i);
  UNTIL ((OfsRacine + i = OfsDonnees - 1) OR (Trouve))
  ELSE
  BEGIN
    Premier := ((Clust*SpC)+OfsDonnees)-(SpC SHL 1);
    i := 0;
    REPEAT
      NoSect := ((Clust * SpC) + OfsDonnees) -
                (SpC SHL 1) + i;
      Cluster := ChercheEntreeFic(Drv, NoSect, Nom);
      IF (Cluster > 0) THEN
        Trouve := True
      ELSE
      BEGIN
        Cluster := ChercheEntreeFAT(Drv, Clust);
        IF (Clust >= Fin) THEN
          Continuer := False
        ELSE
        BEGIN
          Continuer := True;
          Inc(i);
        END;
      END;
    UNTIL ((NOT Continuer) OR (Trouve) OR (i = SpC));
  END;
```

➥

Programme FAT.Pas (suite). ❼

```
  IF Trouve THEN
  BEGIN
    Lig := WhereY + 1;
    GotoXy(18, Lig); Write(Nom);
    GotoXy(32, Lig); Write(Attribut[Entree.Attr OR 1],
                        Attribut[Entree.Attr OR 2],
                        Attribut[Entree.Attr OR 4],
                        Attribut[Entree.Attr OR 16],
                        Attribut[Entree.Attr OR 32]);
    GotoXy(40, Lig); Write(Entree.Taille:7);
    GotoXy(55, Lig); Write(Entree.Cluster:5);
    ChercheRepertoire := Cluster;
  END
  ELSE
    ChercheRepertoire := 0;
END;

FUNCTION ChercheFichier(Drv : Byte; Sect : Word;
                        VAR Nom : String) : Boolean;
VAR Clust  : Word;
    Lig    : Byte;
    Trouve : Boolean;
BEGIN
  Lig := WhereY + 1;
  Clust := ChercheEntreeFic(Drv, Sect, Nom);
  IF (Clust > 0) THEN
  BEGIN
    GotoXy(18, Lig); Write(Nom);
    GotoXy(32, Lig); Write(Attribut[Entree.Attr OR 1],
                          Attribut[Entree.Attr OR 2],
                          Attribut[Entree.Attr OR 4],
                          Attribut[Entree.Attr OR 16],
                          Attribut[Entree.Attr OR 32]);
    GotoXy(40, Lig); Write(Entree.Taille:7);
    REPEAT
      GotoXy(55, Lig); Write(Clust:5); Inc(Lig);
      Clust := ChercheEntreeFAT(Drv, Clust);
      Trouve := True;
    UNTIL (Clust >= Fin);
  END
  ELSE
    Trouve := False;
  ChercheFichier := Trouve;
END;
```

➥

Programme FAT.Pas (suite). ❽

```
 PROCEDURE Ecran;
 BEGIN
   FillChar(Attribut, SizeOf(Attribut), '.');
   Attribut[1] := 'L'; Attribut[2] := 'H';
   Attribut[4] := 'S'; Attribut[8] := 'V';
   Attribut[16] := 'R'; Attribut[32] := 'A';
   Sauve := TextAttr;
   TextAttr := 15+1*16; ClrScr; GotoXy(30, 3);
   Write(Chaine);
   TextAttr := 14+4*16; GotoXy(37, 1); Write(' F A T ');
   GotoXy(5, 3); Write(' Fichier à Analyser : ');
   GotoXy(16, 5);
   Write(
   '   Nom        | Type  |  Taille   | Cluster 1   ');
   TextAttr := 15 + 1 * 16;
 END;

BEGIN
 IF (ParamCount < 1) THEN
 BEGIN
   WriteLn('Format : Lect:\Chemin\NomFic.Ext');
   Halt(1);
 END;
 Chaine := ParamStr(1); Ecran;
 Chaine[1] := Upcase(Chaine[1]);
 Drv := Ord(Chaine[1]) - 65;
 FSplit(Chaine, D, N, E);
 i := 1; Trouve := False; Clust := 0;
 LitSectBoot(Drv);
 IF SeizeBits THEN
   Fin := $FFF8
 ELSE
   Fin := $FF8;
 REPEAT
   Morceau := CheminAChercher(Chaine);
   IF (Morceau = '') THEN
     Exit;
   IF (Morceau <> N + E) THEN
     IF (i = 1) THEN
     BEGIN
       Clust := ChercheRepertoire(Drv, 0, Morceau);
       PrecDir := Clust;
       IF (Clust = 0) THEN
```

➥

Programme FAT.Pas (suite). ❾

```
        BEGIN
          Write(' Mauvais nom de chemin, ou ');
          Write('chemin relatif ');
          Halt(1);
        END;
      END
      ELSE
      BEGIN
        Clust := ChercheRepertoire(Drv, Clust, Morceau);
        PrecDir := Clust;
      END
    ELSE
    IF (Morceau = (N + E)) THEN
    BEGIN
      Fichier := Morceau;
      IF (Clust = 0) THEN
      BEGIN
        Clust := 2;
        NoSect := OfsRacine;
      END
      ELSE
        NoSect := ((Clust*SpC)+OfsDonnees)-(SpC SHL 1);
      Trouve := ChercheFichier(Drv, NoSect, Fichier);
      IF (NOT Trouve) THEN
      BEGIN
        REPEAT
          Clust := ChercheEntreeFAT(Drv, PrecDir);
          PrecDir := Clust; Morceau := N + E;
          Fichier := Morceau;
          NoSect := ((Clust * SpC) + OfsDonnees) -
                    (SpC SHL 1);
          Trouve := ChercheFichier(Drv, NoSect, Fichier);
        UNTIL ((Trouve) OR (Clust = 0));
      END;
    END;
    Inc(i);
  UNTIL Trouve;
  ReadLn;
  TextAttr := Sauve; ClrScr;
END.
```

Table de partition du disque dur

Nécessité fait loi : telle devait être la devise de Microsoft lors des mises à jour successives du DOS. On pense plus particulièrement ici aux fameux problèmes posés au système d'exploitation par la gestion des disques durs. Que fait-on lorsqu'on a un disque de 300 Mo et le DOS Microsoft ou IBM ? On crée une partition DOS de 32 Mo, et une partition étendue que l'on divise en plusieurs lecteurs logiques. Le DOS dispose naturellement d'une structure de données chargée de gérer ces lecteurs (`C:`, `D:`, `E:`, etc.). C'est la table de partition.

Principes de fonctionnement

Plusieurs contraintes ont été posées aux concepteurs de cette structure de données. Tout d'abord, un disque dur doit pouvoir disposer de plusieurs systèmes d'exploitation. Chacun de ces systèmes fait l'objet d'une partition propre, qui doit pouvoir être bootable à l'exclusion des autres. Cela impose que le BIOS soit à même d'interpréter la table, de façon à pouvoir amorcer la bonne partition. Pour que la table soit accessible au BIOS avant la phase d'amorçage, elle doit se trouver dans les secteurs réservés, que le BIOS lit en premier car ils lui indiquent les particularités du disque. Enfin, elle doit contenir un programme permettant son interprétation.

Lorsque le système est lancé sur le disque, le BIOS lit le secteur `1`, cylindre `0`, tête `0`, le charge à l'adresse `0000h:7C00h` et exécute le programme qui s'y trouve. Celui-ci exécute les opérations suivantes :

1. il se recopie ailleurs en mémoire ;
2. il se donne la main ;
3. il lit chacune des quatre entrées de la table, pour vérifier si elle est amorçable ;
4. si une entrée est bootable, il charge le secteur de boot de cette partition en `0000h:7C00h`
5. si le secteur de boot contient la signature `AA55h`, il l'exécute.

Format d'une table de partition

Outre le programme d'interprétation, ses messages d'erreur et sa pile, une table de partition contient quatre entrées (le DOS n'accepte pas plus de partitions) et deux octets de signature (`AA55h`). Une entrée contient 16 octets qui indiquent l'état (bootable ou non bootable) de la partition, son type (DOS 12 bits, DOS 16 bits, étendue, autre), les numéros de tête, secteur et cylindre auquels elle commence et finit, le premier secteur relatif et le nombre de secteurs de la partition.

Format du secteur de partition

Description	Adresse
Programme	0000h
Message d'erreur n° 1	008Bh
Message d'erreur n° 2	00A3h
Message d'erreur n° 3	00C3h
Pile	00DAh
Partition n° 4	01BEh
Partition n° 3	01CEh
Partition n° 2	01DEh
Partition n° 1	01EEh
Signature	01FEh - 01FFh (55AAh)

Format d'une entrée partition

Description	Adresse
Indicateur de boot (80h = oui, 0 = non)	00h
Tête de début de la partition	01h
Secteur et cylindre de début de la partition *(voir codage au chapitre 4)*	01h
Indicateur système 00h = Inconnu 01h = MS-DOS, 12 bits 02h = XENIX 03h = XENIX 04h = MS-DOS, 16 bits 05h = MS-DOS, étendue 06h = Partition > 32 Mo, DOS 4.0 DBh = Concurrent DOS	04h
Tête de fin de la partition	05h
Secteur et cylindre de fin de la partition *(voir codage au chapitre 4)*	06h
Premier secteur relatif	08h
Nombre de secteurs dans la partition	0Ch-0Fh

Tableau 6.17
Format d'une table de partition et de ses entrées.

Théoriquement, le DOS n'est pas obligé de vérifier s'il écrit les entrées de partition les unes à la suite des autres. Il ne peut y avoir qu'une seule partition bootable. Contrairement aux autres champs, qui ont tous une longueur d'un octet, les numéros de secteurs relatifs et le nombre de secteurs de la partition sont exprimés sur quatre octets. Enfin, il faut se méfier des adresses au-dessous de `01BEh` : elles sont justes dans le DOS IBM 3.3 et dans le DOS Microsoft 3.3, mais peut-être pas pour d'autres versions. Seule une séance de travail avec *Norton Utilities* et DEBUG vous permettra de connaître les adresses exactes auxquelles se trouvent les divers éléments d'une table de partition pour votre version du DOS.

Programme du secteur de partition

Nous reproduisons ici le désassemblage du programme et des données que contient la table de partition de notre disque dur.

```
;          Désassemblage de la table de partition          ;
;                   © Microsoft                            ;

Cli                  ; Ints interdites
Xor  Ax,Ax           ; SS := 0000h
Mov  Ss,Ax
Mov  Sp,7C00h        ; SP := 7C00h, SI := 7C00h
Mov  Si,Sp
Push Ax              ; ES := 0000h, DS := 0000h
Pop  Es
Push Ax
Pop  Ds
Sti                  ; Ints autorisées
Cld                  ; Incrémentation des index
Mov  Di,0600h        ; Copier 0000:7C00h en 0000:0600h
Mov  Cx,0100h        ; 512  octets
RepNz                ; Copie
MovsW
Jmp  0000:061Dh      ; = "Jmp Entree", mais dans la copie
                     ; qui se trouve en 0000h:0600h
Entree:
Mov  Si,07BEh        ; Si := Première entrée Table
Mov  Bl,04h          ; Bl := NbEntrees

Prochain:
Cmp  Byte Ptr [Si],80h  ; partition bootable ?
Jz   Bootable:          ; oui, saut
Cmp  Byte Ptr [Si],00h  ; Si pas bootable et pas 0,
Jnz  Erreur1            ; alors Erreur
```

Désassemblage de la table de partition. ❷

```
Add  Si,10h              ; prochaine entrée de la table
Dec  Bl                  ; Recommencer jusqu'à Bl = 0
Jnz  Prochain
Int  18h                 ; Afficher "Pas de périphérique
                         ; disponible", ou donner la main au
                         ; BASIC
Bootable:
Mov  Dx,[Si]             ; DX := 80h
Mov  Cx,[Si+02h]         ; CX := Secteur + Cylindre
Mov  Bp,Si               ; Bp := Adresse Entrée en cours

FinBoucle:
Add  Si,10h              ; Terminer la boucle avant
Dec  Bl                  ; de lire le secteur de Boot
Jz   LitBoot
Cmp  Byte Ptr [Si],00    ; Si oui, boucler
Jz   FinBoucle

Erreur1:                 ; Adresse message d'erreur n° 1
Mov  Si,068Bh

AfficheErreur:
LodsB                    ; Al := DS:SI
Cmp  Al,00h              ; Si Al = 0, terminer
Jz   BoucleSansFin
Push Si                  ; Sauver index
Mov  Bx,0007h            ; Afficher [Al] page 0 en blanc
Mov  Ah,0Eh
Int  10h
Pop  Si                  ; Récupérer index
Jmp  AfficheErreur       ; Boucler

BoucleSansFin:           ; Il n'y a plus qu'à booter sur
Jmp  BoucleSansFin       ; la disquette et à réparer

LitBoot:
Mov  Di,0005h            ; Nombre maximal de répétitions
LitSectBoot:
Mov  Bx,7C00h            ; Charger secteur en 0000:7C00h
Mov  Ax,0201h            ; lire un seul secteur
Push Di                  ; sauver index avant lecture
Int  13h
Pop  Di                  ; récupérer index
Jnb  VerifieBootRec      ; Si CF = 0, vérifier signature
```

➥

Désassemblage de la table de partition. ❸

```
ResetDisque:              ; Sinon, faire un Reset disque
Xor  Ax,Ax
Int  13h
Dec  Di                   ; Décrémenter di, recommencer
Jnz  LitSectBoot

Erreur2:                  ; Erreur de lecture disque
Mov  Si,06A3h             ; Si := Ofs(MsgErr2)
Jmp  AfficheErreur

Erreur3:                  ; Mettre Si à jour sur message
Mov  Si,06C2h             ; d'erreur n° 3

VerifieBootRec:
Mov  Di,7DFEh             ; Dernier Mot = Signature ?
Cmp  Word Ptr [Di], AA55h
Jnz  AfficheErreur        ; Non, Afficher erreur n° 3

DonneLaMain:              ; Oui, exécuter secteur Boot
Mov  Si,Bp
Jmp  0000:7C00h

Db 'Invalid partition Table',00h        ; Message n° 1
Db 'Error loading operating system',00h ; Message n° 2
Db 'Missing operating system',00h       ; Message n° 3
Db 227 Dup (00h)                        ; Pile

; Entrées de la table de partition
                                   ; Entrée n° 4
Db 80h 01h 01h 00h 04h 07h 51h E0h 11h 00h 00h 00h 77h
Db FFh 00h 00h
                                   ; Entrée n° 3
Db 00h 00h 41h E1h 05h 07h D1h FEh 88h FFh 00h 00h F0h
Db 1Fh 01h 00h
                                   ; Entrée n° 2
Db 00h 00h 00h 00h 00h 00h 00h 00h 00h 00h 00h 00h 00h
Db 00h 00h 00h
                                   ; Entrée n° 1
Db 00h 00h 00h 00h 00h 00h 00h 00h 00h 00h 00h 00h 00h
Db 00h 00h 00h

; Signature du secteur de partition

Db 55h AAh
```

Figure 6.18
Le programme et les données de la table des partitions.

Problème des lecteurs logiques

Les lecteurs logiques sont des subdivisions des partitions DOS étendues. Lorsque l'on crée une partition étendue avec `FDisk` et que celle-ci est d'une taille supérieure à 32 Mo, on doit la diviser en plusieurs lecteurs logiques. A ce moment, la partition étendue est elle-même divisée en une partition DOS et une partition DOS étendue, qui contient le second lecteur logique créé. Celui-ci possède également sa propre table de partition. Bref, on se retrouve, ici comme ailleurs, avec une organisation ressemblant fort à celle des pointeurs. Heureusement, chaque partition indique ses coordonnées de début et de fin, ce qui permet, lorsque l'on a repéré une partition étendue, d'aller lire sa table des partitions, qui se trouve dans le premier secteur physique indiqué par l'entrée à laquelle elle correspond. Il faut toutefois bien retenir que seule la première partition DOS non étendue est bootable.

Le programme `SectPart.Pas` lit les différentes tables de partition existantes du disque fixe numéro 1 et en affiche les valeurs.

Nom	*Type*	*Description*	*Ligne*
`Reset`	F	Initialise le disque	*10*
`LitSectBios`	F	Lit un secteur du disque en passant par la fonction `02h` de l'Int `13h`	*24*
`Etendue`	F	Renvoie l'adresse de la partition étendue s'il y en a une	*45*
`AffPartition`	P	Affiche une entrée de partition	*60*
`Affiche`	P	Affiche les 4 entrées d'une table de partition	*94*
`Ecran`	P	Dessine l'écran	*105*
`TrouveTables`	P	Cherche toutes les tables de partition d'un disque	*116*

Dépendances :

Procédure	*Appelle procédure*
`Programme`	`Ecran`
	`TrouveTables`
`TrouveTables`	`Reset`
	`LitSectBios`
	`Etendue`
	`Affiche`
`Affiche`	`AffPartition`

Tableau 6.19
Références croisées de SectPart.Pas.

Listing 6.20
Programme SectPart.Pas. ❶

```
PROGRAM LitSecteurDePartition;    { SectPart.Pas }

USES Dos, Crt, Sys;

TYPE Tableau   = ARRAY[0..511] OF Byte;

VAR TabSect    : Tableau;
    Sauve      : Byte;

 FUNCTION Reset : Byte;
 VAR Regs : Registers;
 BEGIN
   WITH Regs DO
   BEGIN
     Ah := 0;
     Intr($13, Regs);
     IF (Flags AND 1 = 1) THEN
       Reset := Ah
     ELSE
       Reset := 0;
   END;
 END;

FUNCTION LitSectBios(Drv, NoSect, NbSect, Tete : Byte;
                     Cyl : Word) : Byte;
 VAR Regs : Registers;
 BEGIN
   WITH Regs DO
   BEGIN
     Ah := 2;
     Al := NbSect;
     Ch := Lo(Cyl);
     Cl := (Hi(Cyl) AND $C0) + NoSect;
     Dh := Tete;
     Dl := Drv;
     Es := Seg(TabSect);
     Bx := Ofs(TabSect);
     Intr($13, Regs);
     IF (Flags AND 1 = 1) THEN
       LitSectBios := Ah
     ELSE
       LitSectBios := 0;
   END;
 END;
```

➥

Programme SectPart.Pas. ❷

```
FUNCTION ChercheEtendue : Word;
VAR Indice : Word;
BEGIN
  Indice := $01AE;
  {$R-}
  REPEAT
    Inc(Indice, $10);
  UNTIL ((Indice = $01FE) OR (TabSect[Indice + 4] = 5));
  {R+}
  IF (Indice <> $01FE) THEN
    ChercheEtendue := Indice
  ELSE
    ChercheEtendue := 0;
END;

PROCEDURE AffichePartition(No : Word; Lig : Byte);
VAR Indice, Debut, Nb : LongInt;
 BEGIN
   GotoXy(3, Lig);
   Write(Boole(TabSect[No] AND $80 SHR 7));
   GotoXy(14, Lig); Write(TabSect[No + 1]);
   CylEtSect := TabSect[No + 3];
   CylEtSect := CylEtSect SHL 8;
   CylEtSect := CylEtSect + TabSect[No + 2];
   Cylindre := Lo(CylEtSect) SHR 6;
   Cylindre := Cylindre SHL 8;
   Cylindre := Cylindre + (Hi(CylEtSect));
   Secteur  := Lo(CylEtSect) AND $3F;
   GotoXy(21, Lig); Write(OctetDecVersHex(Secteur), 'h');
   GotoXy(29, Lig); Write(MotDecVersHex(Cylindre), 'h');
   GotoXy(36, Lig);
   IF TabSect[No + 4] = 1 THEN
     Write('12 b')
   ELSE IF TabSect[No + 4] = 4 THEN
     Write('16 b')
   ELSE IF TabSect[No + 4] = 5 THEN
     Write('EXT ')
   ELSE
     Write(' ?  ');
   GotoXy(43, Lig); Write(TabSect[No + 5]);
   CylEtSect := TabSect[No + 7];
   CylEtSect := CylEtSect SHL 8;
   CylEtSect := CylEtSect + TabSect[No + 6];
   Cylindre := Lo(CylEtSect) SHR 6;
   Cylindre := Cylindre SHL 8;
```

➥

Programme SectPart.Pas. ❸

```
    Cylindre := Cylindre + Hi(CylEtSect);
    Secteur  := Lo(CylEtSect) AND $3F;
    GotoXy(50, Lig);
    Write(OctetDecVersHex(Secteur), 'h');
    GotoXy(58, Lig);
    Write(MotDecVersHex(Cylindre), 'h');
    Indice := TabSect[No + $B];
    Debut := (Indice SHL 24);
    Indice := TabSect[No + $A];
    Debut := Debut + LongInt(Indice SHL 16);
    Indice := TabSect[No + 9];
    Debut := Debut + LongInt(Indice SHL 8);
   Indice := TabSect[No + 8];
   Debut := Debut + LongInt(Indice);
   GotoXy(64, Lig); Write(Debut:7);
   Indice := TabSect[No + $F];
   Nb := (Indice SHL 24);
   Indice := TabSect[No + $E];
   Nb := Nb + LongInt(Indice SHL 16);
   Indice := TabSect[No + $D];
   Nb := Nb + LongInt(Indice SHL 8);
   Indice := TabSect[No + $C];
   Nb := Nb + LongInt(Indice);
   GotoXy(72, Lig); Write(Nb:7);
 END;

 PROCEDURE Affiche(Lig : Byte);
 BEGIN
   TextAttr := 15 + 1 * 16;
   AffichePartition($01BE, Lig + 1);
   AffichePartition($01CE, Lig + 2);
   AffichePartition($01DE, Lig + 3);
   AffichePartition($01EE, Lig + 4);
   GotoXy(2, Lig + 5);
   Write('------------------------------------------');
   Write('---------------------------------');
 END;

 PROCEDURE Ecran;
 BEGIN
   Sauve := TextAttr;
   TextAttr := 15+1*16; ClrScr; TextAttr := 14+4*16;
   GotoXy(20, 1);
   Write(' É D I T E U R   de   P A R T I T I O N S ');
```

➥

Programme SectPart.Pas. ❹

```
    GotoXy(2, 4);
    Write('                Début disque                ');
    Write('   Fin Disque       Sect. relatifs ');
    GotoXy(2, 5);
    Write(' Bootable | Tête | Sect. | Cyl. | Type | ');
    Write('Tête | Sect. | Cyl. | 1° sect | Nbre ');
  END;

  PROCEDURE TrouveLesTables;
  VAR Tete, Sect,
      i, Lig, Erreur : Byte;
      Cyl, No        : Word;
  BEGIN
    Tete := 0; Sect := 1; Cyl := 0; i := 1; Lig := 6;
    REPEAT
      Erreur := Reset; Inc(i);
    UNTIL ((Erreur = 0) OR (i = 3));
    IF (Erreur = 0) THEN
      Erreur := LitSectBios($80, Sect, 1, Tete, Cyl);
    Affiche(Lig);
    REPEAT
      No := ChercheEtendue;
      IF (No <> 0) THEN
      BEGIN
        Tete := TabSect[No + 1]; Sect := TabSect[No + 2];
        Cyl := TabSect[No + 3]; i := 1;
        REPEAT
          Erreur := Reset; Inc(i);
        UNTIL ((Erreur = 0) OR (i = 3));
        IF (Erreur = 0) THEN
        BEGIN
          Erreur := LitSectBios($80, Sect, 1, Tete, Cyl);
          Inc(Lig, 6);
          Affiche(Lig);
        END;
      END;
    UNTIL (No = 0);
  END;

BEGIN
  FillChar(TabSect, SizeOf(TabSect), 0);
  Ecran;
  TrouveLesTables;
  ReadLn;
  TextAttr := Sauve; ClrScr;
END.
```

Conclusion

Nous avons vu toutes les structures DOS de bas niveau concernant le disque : secteur de boot, FAT et table de partition. Nous avons désassemblé toutes celles qui pouvaient l'être et indiqué leur rôle, leur fonctionnement et la signification de leurs données. Les programmes exemples de ce chapitre devraient également vous avoir aidé à comprendre ces éléments principaux de la gestion des disques.

Il nous reste encore à voir les structures de haut niveau, après quoi le disque ne devrait (presque) plus avoir de secrets pour vous.

Chapitre 7

Disques au niveau logique : structures DOS de haut niveau

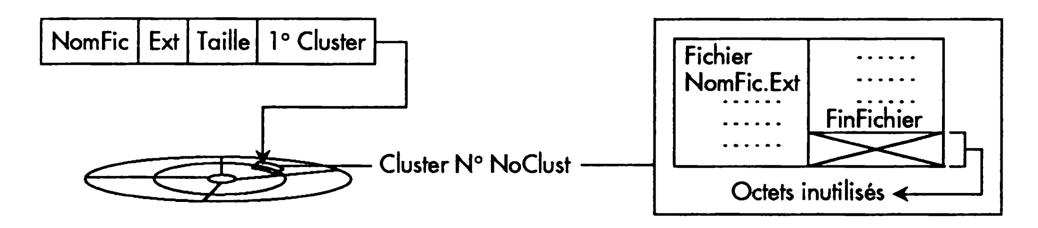
Mots-clefs

Entrées fichiers	On les appelle aussi en-têtes de fichiers. Il s'agit de la structure mise en place par le DOS pour lui indiquer les principales caractéristiques d'un fichier : taille, date et heure de création, emplacement sur le disque.
Fragmentation	On dit qu'un disque est fragmenté lorsque de nombreux blocs libres apparaissent éparpillés entre des blocs occupés. Un disque fragmenté ralentit les opérations du DOS sur les fichiers.
Répertoire racine	Le répertoire racine, situé juste après la seconde copie de la FAT sur le disque, ne contient que des entrées fichiers, comme tout répertoire, mais a une taille limitée.

Ce chapitre complète le précédent et en éclaire certains points. Après nous être intéressés aux entrées-fichiers, au répertoire racine et à la zone des fichiers, nous serons en mesure de comprendre de quelle façon les structures de haut et de bas niveaux travaillent ensemble.

Entrées fichiers

Le DOS ne traite pas les fichiers comme des ensembles homogènes. Un fichier se divise pour lui en deux parties : un en-tête et une zone d'information proprement dite. La zone d'information peut elle-même se trouver fragmentée en plusieurs morceaux qu'il faudra recoller lors d'une mise à jour. Cette façon de faire procure plusieurs avantages. On peut réserver certains espaces du disque aux en-têtes, de manière à les retrouver facilement. La fragmentation de la zone d'information permet une mise à jour aisée : sans elle, le système serait obligé de recopier l'intégralité du fichier à un autre endroit du disque à chaque ajout de données. L'espace disque serait si fragmenté qu'il contiendrait moins de fichiers.

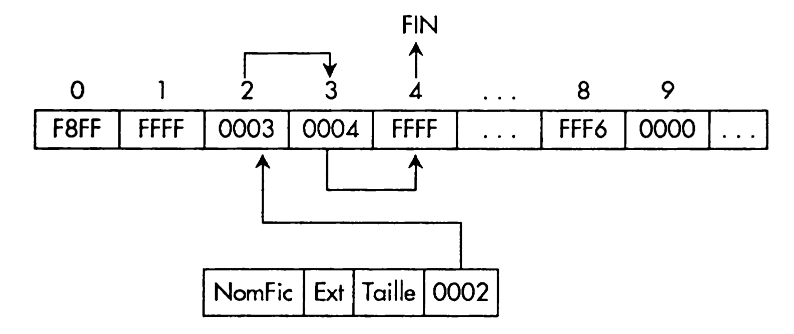


Une entrée fichier contient un champ indiquant le numéro du cluster où se trouve le fichier.

Figure 7.1
Les deux parties d'un fichier.

Principes de fonctionnement

L'en-tête joue ici un rôle fondamental : comme il fait partie des rares structures de données dont on connaît l'espacement à l'avance, il est facilement accessible et contient d'importantes informations sur le fichier dont il est responsable. C'est pourquoi on appelle cet en-tête une *entrée fichier*. Parmi les renseignements qu'elle procure au DOS, une entrée-fichier désigne le premier cluster auquel se trouve le fichier. Les suivants peuvent être obtenus en lisant la FAT à l'index de même numéro que le cluster (voir le chapitre 6, *Disques au niveau logique : structures DOS de bas niveau*).



Les entrées fichiers sont indispensables pour interpréter la FAT, qui permet de localiser le fichier en son entier.

Figure 7.2
Liaisons FAT-Entrées fichiers.

Format des entrées fichiers

Une entrée fichier contient trente-deux octets. En dehors du nom et de l'extension du fichier, elle renseigne le système d'exploitation sur son attribut, sa date et son heure de création, sa taille et – surtout – sur son adresse.

Adresse	*Signification*
00h	Nom du fichier Premier octet = 00h : Plus d'entrées dans le cluster E5h : Fichier effacé 05h : Caractère E5h
08h	Extension
0Bh	Attribut Bit 0 = 1 Lecture seule; 4 = 1 Répertoire 1 = 1 Caché; 5 = 1 Archive 2 = 1 Système; 6 Réservé 3 = 1 Volume; 7 Réservé
0Ch	Réservé
16h	Heure de création ou de mise à jour
18h	Date de création ou de mise à jour
1Ah	Cluster de début du fichier
1Ch-1Fh	Taille du fichier

Tableau 7.3
Format d'une entrée fichier.

Les bits 6 et 7 de l'octet 0Bh (attribut) sont inutilisés. Différents attributs peuvent être combinés. Les octets 0Ch à 15h sont inutilisés et généralement à zéro. L'heure et la date sont codés.

```
Heure1 := Entree[$17] SHL 8 + Entree[$16]; {Heure et Date }
Date1  := Entree[$19] SHL 8 + Entree[$18]; {sont des WORD }
                         { Entree est un tableau d'octets }
H     := (Heure1 SHR $B);         { Décodage }
Mn    := (Heure1 SHR 5) AND $3F;
Sec   := (Heure1 AND $1F) SHL 1;
Annee := (Date1 SHR 9) + 1980;
Mois  := (Date1 SHR 5) AND $F;
Jour  := (Date1 AND $1F);
                                 { Codage }
Heure2 := (H SHL $B) + (Mn SHL 5) + (Sec SHR 1);
Date2 := ((Anne-1980) SHL 9) + (Mois SHL 5) + Jour;

IF ((Heure1 = Heure2) AND (Date1 = Date2)) THEN
   Ok := TRUE;                           { Vérification }
```

Encadré 7.4
Décoder la date et l'heure d'un fichier.

Remarque — Il est à noter que le Turbo Pascal considère l'ensemble Heure + Date comme un entier long (voir ce que cela implique dans le petit programme d'exemple *listing 7.1*).

Le programme d'exemple qui suit montre comment on décode la date et l'heure d'un fichier en Turbo Pascal, mais emploie une procédure prédéfinie de l'unité DOS pour les obtenir.

Listing 7.5
Afficher l'heure et la date d'un fichier : Programme HeureFic.Pas. ❶

```
PROGRAM TrouveHeureFichier; { HeureFic.Pas }

USES Dos;

VAR  Fichier : FILE;
     NomFic  : STRING;

PROCEDURE OuvreFichier(VAR Fic : FILE; Nom : STRING);
BEGIN                { La procédure GetFTime impose que }
  Assign(Fic, Nom);  { le fichier ait été ouvert }
  {$I-}
  Reset(Fic);
  {$I+}
```

➡

Programme HeureFic.Pas (suite). ❷

```
  IF (IOResult <> 0) THEN
  BEGIN
    WriteLn('Erreur en ouverture de fichier');
    Halt;
  END;
END;

PROCEDURE AffHeureEtDate(VAR Fic : FILE; Nom : STRING);
CONST Mois : ARRAY[1..12] OF STRING[8] =
             ('Janvier', 'Février', 'Mars',
             'Avril', 'Mai', 'Juin',
             'Juillet', 'Août', 'Septembre',
             'Octobre', 'Novembre', 'Décembre');
VAR   HeureEtDate          : LongInt;
      H, Mn, S, J, M, An   : Word;
BEGIN
  GetFTime(Fic, HeureEtDate);
  Close(Fic);
  Heure := HeureEtDate AND $0000FFFF;
  Date  := (HeureEtDate AND $FFFF0000) SHR $10;
  H     := Heure SHR $B;
  Mn    := (Heure SHR 5) AND $3F;
  S     := (Heure AND $1F) SHL 1;
  An    := (Date SHR 9) + 1980;
  M     := (Date SHR 5) AND $F
  J     := Date AND $1F;
  WriteLn('Fichier ', Nom, ' créé le : ');
  Write(J, ' ', Mois[M], ' ', An);
  WriteLn(' à ', H, ' h, ', Mn, ' mn, ', S, ' s');
END;

BEGIN
  IF (ParamCount < 1) THEN
  BEGIN
    WriteLn(' Nom de fichier en ligne de commande !');
    Halt;
  END
  ELSE
  BEGIN
    NomFic := ParamStr(1);
    OuvreFichier(Fichier, NomFic);
    AffHeureEtDate(Fichier, NomFic);
  END;
```

Une entrée fichier désigne aussi bien un fichier qu'un répertoire ou un label de volume. S'il s'agit d'un répertoire, sa taille est à zéro. Son nom peut être composé d'un ou de deux points. Si son nom est «..   », il désigne son répertoire père et contient le numéro de cluster auquel celui-ci commence. Si son nom est «.   », il se désigne lui-même et contient son propre numéro de cluster. Cette particularité permet au DOS de gérer les chemins d'accès relatifs. Les labels de volume, quant à eux, ont leur champ taille et leur champ cluster à zéro. Il ne peut y avoir qu'un seul label de volume par disque.

Répertoires et entrées fichiers

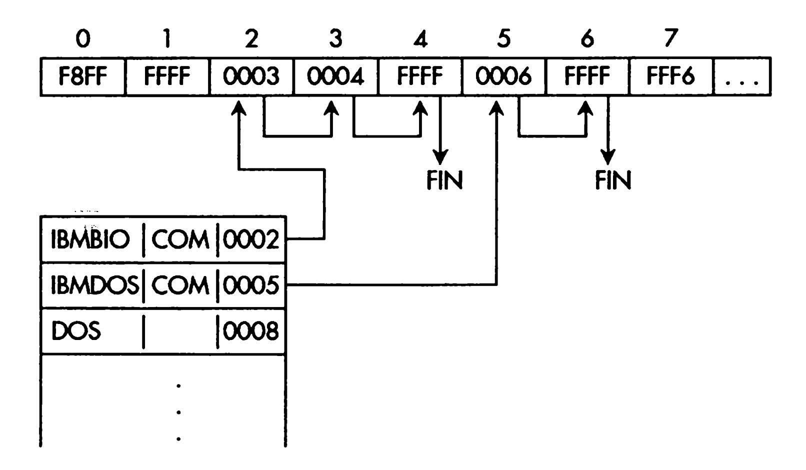
Il nous faut distinguer entre deux catégories de répertoires, avant de pouvoir examiner les liens qui existent entre eux et les entrées fichiers. Il s'agit du répertoire racine et des sous-répertoires.

Répertoire racine et entrées fichiers

Le répertoire racine a ceci de particulier qu'il se trouve à un emplacement fixe du disque et que l'on ne peut pas l'effacer. Il vient juste après le secteur de boot et la FAT, et comporte de 4 à 32 secteurs selon les formats de disque ou de disquette (voir les données du secteur de boot au chapitre 6, *Disques au niveau logique : structures DOS de bas niveau*).

Ces secteurs et les informations qu'ils contiennent ne sont pas répertoriés par la FAT. La racine ne contient en effet que des entrées fichiers, qui pointent soit sur des fichiers soit sur des répertoires. La racine est donc la seule source d'informations du DOS concernant la structure du disque. Comme sa taille est fixée au moment du formatage logique, le nombre d'entrées fichiers du répertoire racine est forcément limité. Il vaut donc mieux que ces entrées pointent sur des répertoires plutôt que sur des fichiers.

On peut voir la racine du disque comme une table de hachage : un tableau unidimensionnel dont chaque enregistrement serait un pointeur lié à la FAT. Tant qu'on en reste aux fichiers, il s'agit d'une structure relativement simple (voir *figure 7.6*). Dès qu'on y ajoute les sous-répertoires, cela se complique énormément.



Le répertoire racine vu comme une table de hachage.

Figure 7.6
La racine : une table de hachage.

Sous-répertoires et entrées fichiers

Nous avons dit tout à l'heure qu'une entrée fichier pouvait aussi bien pointer sur un fichier que sur un répertoire, et qu'il valait mieux que la racine soit composée de sous-répertoires plutôt que de fichiers. Nous allons maintenant expliquer pourquoi.

Un sous-répertoire se comporte exactement comme le répertoire racine : c'est-à-dire qu'il contient lui aussi un ensemble d'entrées fichiers, qui peuvent elles-mêmes pointer sur des répertoires. Mais au contraire de la racine, il est référencé par la FAT (puisqu'il est pointé par une entrée fichier) et peut par conséquent être étendu. Ainsi, alors que la racine d'un disque de 32 Mo comporte un maximum de 512 entrées fichiers, un sous-répertoire du même disque peut (théoriquement) contenir autant d'entrées fichiers qu'il y a de clusters sur le disque moins ceux que lui-même occupe, c'est-à-dire un peu plus de 14 000 :

```
(16 000 - ((((16 000 * 512) / 32) / 512)*4))...
```

Si une organisation hiérarchisée est toujours la bienvenue en informatique, où tout tend à se mélanger si vite, celle-ci n'est sans doute pas la meilleure que l'on aurait pu trouver. Chaque répertoire fonctionnant selon les mêmes principes que la racine, on se retrouve en effet dans la situation que décrit la *figure* 7.7.

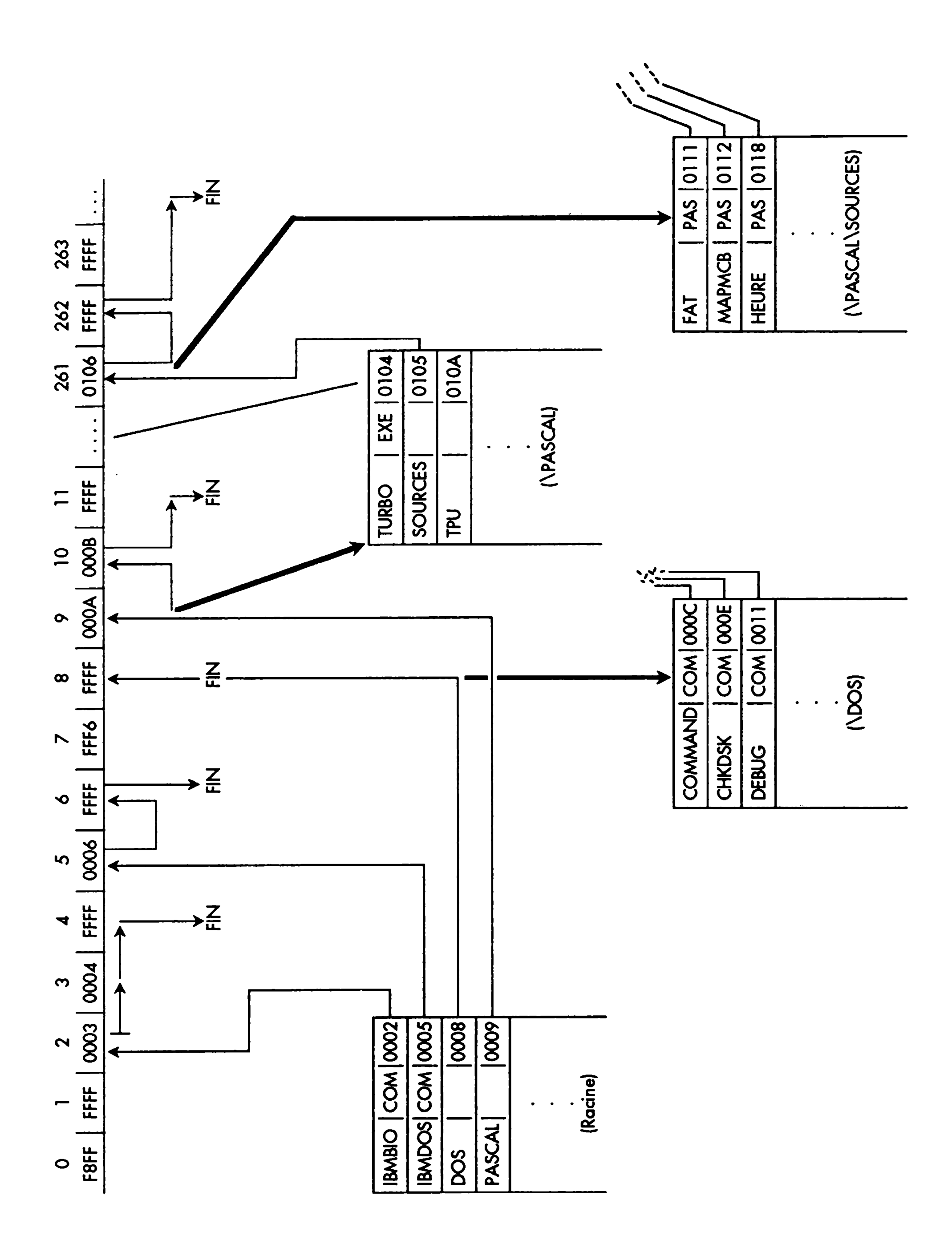


Figure 7.7
Organisation du disque en termes de structures.

Naviguer dans les entrées fichiers

Le programme `Entrees.Pas` a pour objectifs :

1. de visualiser la structure des entrées fichiers et des répertoires ;
2. de comprendre le fonctionnement de ces structures ;
3. de mettre en lumière leurs relations avec la FAT ;
4. de permettre la navigation parmi les entrées fichiers des répertoires.

Ce programme est long (près de 500 lignes). Il y a deux raisons à cela :

1. des tests particulièrement nombreux (dans quel cluster se trouve-t-on, dans quel secteur à l'intérieur du cluster, y a-t-il des entrées fichiers dans le secteur suivant, etc.) ;
2. la gestion des numéros de cluster et de secteurs, qui a nécessité la création d'une pile de mots générant ainsi de nouveaux tests.

Procédure	*Type*	*Description*	*Ligne*
`LecteurCourant`	F	Renvoie le numéro du lecteur en cours	*34*
`Init`	P	Lit le secteur de boot et initialise des variables globales	*45*
`LitFAT`	F	Renvoie le numéro du cluster suivant	*69*
`DernierSectRac`	F	Renvoie le dernier secteur occupé dans la racine	*94*
`Ascii`	F	Renvoie une chaîne copiée dans une suite d'octets numériques	*120*
`LitRep`	P	Lit le secteur 1 du premier cluster d'un répertoire et interprète les entrées qu'il contient	*128*
`Decode`	F	Renvoie une chaîne contenant l'heure et la date d'une entrée	*134*
`\Zéro`	F	Justifie une chaîne de 2 chiffres à l'aide de zéros	*138*
`EcranRep`	P	Affiche l'écran	*198*
`AfficheEntree`	P	Affiche une entrée	*208*
`Attribut`	F	Renvoie une chaîne composée de points et des initiales des attributs d'une entrée	*210*
`AffRep`	P	Affiche un secteur d'entrées fichiers	*240*
`LitClavRep`	P	Lit le clavier, déplace le curseur, passe d'un répertoire à un autre : la procédure principale du programme (et la plus longue)	*268*

➥

(suite du tableau)

Procédure	**Type**	**Description**	**Ligne**
Suivant	F	Vérifie si le prochain secteur est vide ou non	*276*
Espaces	F	Renvoie le nom de chemin vidé de ses espaces inutiles	*282*
\VireEspFin	F	Supprime les espaces de fin d'une chaîne	*285*
VireRep	F	Supprime le nom de répertoire final du chemin	*310*
Push	P	Empile un mot	*320*
Pop	F	Dépile un mot	*335*
RAZPile	P	Met la pile à zéro	*348*
Programme principal			*508*

Dépendances :

Procédure	**Appelle procédure**
LecteurCourant	
Init	LitSectAbs
DernierSectRac	LitSectAbs
LitRep	LitSectAbs
	Ascii
	\Decode
Decode	\Decode\Zero
EcranRep	
AffRep	
	AfficheEntree
AfficheEntree	AfficheEntree\Attribut
LitFAT	LitSectAbs
LitClavRep	LitFAT
	LitRep
	EcranRep
	AfficheEntree
	AffRep
	\Suivant

(suite du tableau)

Procédure	Appelle procédure
LitClavRep *(suite)*	\Espaces
	\VireRep
	\Push
	\Pop
	\RAZPile
Suivant	LitSectAbs
Espaces	\Espaces\VireEspFin
Programme principal	LecteurCourant
	Init
	DernierSectRac
	LitRep
	EcranRep
	AffRep
	LitClavRep

Tableau 7.8
Références croisées de Entrees.Pas.

Listing 7.9
Programme Entrees.Pas. ❶

```
PROGRAM AfficheEntreesFic; { Entrees.Pas }

USES Dos, Crt;

TYPE
 OcPtr = ^Byte;
 Entree = RECORD
            Nom    : STRING[8];
            Ext    : STRING[3];
            Date,
            Heure  : STRING[10];
            Attr   : Byte;
            Taille : LongInt;
            Clust1 : Word;
          END;
 TabEntr = ARRAY[0..16] OF Entree;
 TabSect = ARRAY[0..511] OF Byte;
```

➥

Programme Entrees.Pas (suite). ❷

```
VAR
 TabRep, TabFAT, TabAuxRep             : TabSect;
 TabFic                                : TabEntr;
 Drv, Sauve                            : Byte;
 Secteur, OpS, Fin, SpC, OfsDonnees,
 OfsRacine, NbS, FinRacine             : Word;
 SeizeBits                             : BOOLEAN;
 Chemin, Lect                          : STRING;

 {$L Absolute.Obj }
 {$F+}
 PROCEDURE LitSectAbs(Drv : Byte; No, Nb : Word;
                      Tab : OcPtr); EXTERNAL;
 PROCEDURE EcritSectAbs(Drv : Byte; No, Nb : Word;
                        Tab : OcPtr); EXTERNAL;
 {$F-}

 FUNCTION LecteurCourant : Byte;
 VAR Regs : Registers;
 BEGIN
   Regs.Ah := $19;
   MsDos(Regs);
   IF (Regs.Flags AND 1 = 1) THEN
     LecteurCourant := $F
   ELSE
     LecteurCourant := Regs.Al
 END;

 PROCEDURE Init(Drv : Byte);
 VAR Srv, SC, SpF, NbE : Word;
     NbF               : Byte;
 BEGIN
   LitSectAbs(Drv, 0, 1, @TabRep);
   OpS := (TabRep[$C] SHL 8) + TabRep[$B];
   SpC := TabRep[$D];
   Srv := (TabRep[$F] SHL 8) + TabRep[$E];

   NbF := TabRep[$10];
   NbE := (TabRep[$12] SHL 8) + TabRep[$11];
   NbS := (TabRep[$14] SHL 8) + TabRep[$13];
   SpF := (TabRep[$17] SHL 8) + TabRep[$16];
   SC := (TabRep[$1F] SHL 24) +
         (TabRep[$1E] SHL 16) +
         (TabRep[$1D] SHL 8) + TabRep[$1C];
```

➥

Programme Entrees.Pas (suite). ❸

```
    SeizeBits := ((NbS DIV SpC) > 4078);
    IF SeizeBits THEN
      Fin := $FFF8
    ELSE
      Fin := $FF8;
    OfsRacine := ((NbF * SpF) + Srv);
    OfsDonnees := OfsRacine + ((NbE SHL 5) DIV OpS);
    FillChar(TabRep, SizeOf(TabRep), 0);
  END;

  FUNCTION LitFAT(Drv : Byte; Clust : Word) : Word;
  VAR Sect, Idx, OfsF, Res : Word;
  BEGIN
    Res := 0;
    IF SeizeBits THEN
    BEGIN
      Sect := (Clust DIV (OpS SHR 1)) + 1;
      LitSectAbs(Drv, Sect, 1, @TabFAT);
      Idx := (Clust MOD (OpS SHR 1)) SHL 1;
      Res := (TabFAT[Idx + 1] SHL 8) + TabFAT[Idx];
    END
    ELSE
    BEGIN
      OfsF := Clust + (Clust SHR 1);
      Sect := (OfsF DIV OpS) + 1;
      LitSectAbs(Drv, Sect, 1, @TabFAT);
      Idx := OfsF MOD OpS;
      Res := (TabFAT[Idx + 1] SHL 8) + TabFAT[Idx];
      IF (Clust MOD 2 = 0) THEN
        Res := Res AND $0FFF
      ELSE
        Res := Res SHR 4;
    END;
    LitFAT := Res;
  END;

  FUNCTION DernierSectRac(Drv : Byte) : Word;
  VAR i, SC : Word;
      Fini  : BOOLEAN;
  BEGIN
    Fini := FALSE; SC := OfsRacine; i := 0;
    WHILE ((SC < OfsDonnees) AND (NOT(Fini))) DO
    BEGIN
      LitSectAbs(Drv, SC, 1, @TabRep); i := 0;
```

➥

Programme Entrees.Pas (suite). ❹

```
     REPEAT
        IF (TabRep[i] <> 0) THEN
        Inc(i, 32);
     UNTIL ((TabRep[i] = 0) OR (i = (512 - 32)));
     IF ((i <= (512 - 32)) AND (TabRep[i] = 0)) THEN
       Fini := TRUE
     ELSE
       Inc(SC);
   END;
   IF Fini THEN
     IF (i > 0) THEN
       DernierSectRac := SC
     ELSE
       DernierSectRac := SC - 1
   ELSE
     DernierSectRac := OfsDonnees - 1;
 END;

 FUNCTION Ascii(VAR Tab; Lng : Word) : STRING;
 VAR Chaine : STRING;
 BEGIN
   Chaine[0] := Chr(Lng); FillChar(Chaine[1], Lng, ' ');
   Move(Mem[Seg(Tab):Ofs(Tab)], Chaine[1], Lng);
   Ascii := Chaine;
 END;

 PROCEDURE LitRep(Drv : Byte; Sect : Word);
 VAR i              : Byte;
     j              : Word;
     Chaine         : STRING;
     TailFic, HetD  : LongInt;

   FUNCTION Decode(Heure : LongInt) : STRING;
   VAR Chaine, S : STRING;
       RegD      : DateTime;

     FUNCTION Zero(Chaine : STRING) : STRING;
     BEGIN
       IF (Length(Chaine) < 2) THEN
         Zero := '0' + Chaine
       ELSE
         Zero := Chaine;
     END;
```

➡

Programme Entrees.Pas (suite). ❺

```
    BEGIN { Decode }
      Chaine := ''; UnPackTime(Heure, RegD);
      WITH RegD DO
      BEGIN
        Str(Day, S); Chaine := Zero(S) + '/';
        Str(Month, S); Chaine := Chaine + Zero(S) + '/';
        Str(Year, S); Chaine := Chaine + Zero(S) + '  ';
        Str(Hour, S); Chaine := Chaine + Zero(S) + ':';
        Str(Min, S); Chaine := Chaine + Zero(S) + ':';
        Str(Sec, S); Chaine := Chaine + Zero(S);
      END;
      Decode := Chaine;
    END; { Decode }

  BEGIN { LitRep }
    LitSectAbs(Drv, Sect, 1, @TabRep);
    WITH TabFic[0] DO
    BEGIN
      Nom := ''; Ext := ''; Attr := 0; TailFic := 0;
      Date := ''; Heure := ''; Clust1 := 0; Taille := 0;
    END;
    i := 1; j := 0;
    REPEAT
      WITH TabFic[i] DO
      BEGIN
        Nom := Ascii(TabRep[j], 8);
        Ext := Ascii(TabRep[j + 8], 3);
        Attr := TabRep[j + $B];
        TailFic := (TabRep[j + $19]);
        TailFic := TailFic SHL 24;
        HetD := TailFic;
        TailFic := (TabRep[j + $18]);
        TailFic := TailFic SHL 16;
        HetD := HetD + TailFic;
        TailFic := (TabRep[j + $17]);
        TailFic := TailFic SHL 8;
        HetD := HetD + TailFic;
        HetD := HetD + TabRep[j + $16];
        Chaine := Decode(HetD);
        Date := Copy(Chaine, 1, Pos(' ', Chaine));
        Heure := Copy(Chaine, Pos(' ', Chaine) + 2,
                      Length(Chaine));
        Clust1 := (TabRep[j + $1B] SHL 8) +
                   TabRep[j + $1A];
```

➥

Programme Entrees.Pas (suite). ❻

```
        TailFic := TabRep[j + $1F];
        TailFic := TailFic SHL 24;
        Taille  := TailFic; TailFic := TabRep[j + $1E];
        TailFic := TailFic SHL 16;
        Taille := Taille + TailFic;
        TailFic := TabRep[j + $1D];
        TailFic := TailFic SHL 8;
        Taille := Taille + TailFic;
        Taille := Taille + TabRep[j + $1C];
      END;
      Inc(i); Inc(j, 32);
    UNTIL ((i > 16) OR (TabRep[j - 32] = 0));
    IF (TabRep[j - 32] = 0) THEN
      TabFic[0].Attr := (j - 1) DIV 32
    ELSE
      TabFic[0].Attr := 16;
  END; { LitRep }

  PROCEDURE EcranRep;
  BEGIN
    TextAttr := 15 + 1 * 16; ClrScr; TextAttr := 14+4*16;
    GotoXy(31,1); Write(' Entrées fichiers ');
    GotoXy(1,1);
    Write(' Cluster n° '); GotoXy(18,1); Write(' Sect ');
    GotoXy(67, 1); Write(' '); GotoXy(70, 1);
    Write(' entrées');
    TextAttr := 4 + 7 * 16; GotoXy(20, 3); Write('':40);
    TextAttr := 15 + 1 * 16;
  END;

  PROCEDURE AfficheEntree(Fic : Entree; No, Attr : Byte);

    FUNCTION Attribut(Attr : Byte) : STRING;
    VAR Chaine : STRING[6];
    BEGIN
      Chaine := '......';
      IF (Attr OR 1 = Attr) THEN
        Chaine[1] := 'H';
      IF (Attr OR 2 = Attr) THEN
        Chaine[2] := 'S';
      IF (Attr OR 4 = Attr) THEN
        Chaine[3] := 'R';
      IF (Attr OR 8 = Attr) THEN
        Chaine[4] := 'V';
```

➥

Programme Entrees.Pas (suite). ❼

```
    IF (Attr OR $10 = Attr) THEN
      Chaine[5] := 'D';
    IF (Attr OR $20 = Attr) THEN
      Chaine[6] := 'A';
    Attribut := Chaine;
  END;
BEGIN { AfficheEntree}
  GotoXy(4, 6 + No); TextAttr := Attr; Write('':70);
  GotoXy(5, 6 + No);
  WITH Fic DO
  BEGIN
    Write(' ', Nom, ' ', Ext, '':4);
    Write(Attribut(Attr), '':4,Date,'':4,Heure, ' ');
    Write(Taille:8, '':4, Clust1:5);
  END;
END; { AfficheEntree}

PROCEDURE AffRep(Clust, Sect : Word);
VAR i, Max : Byte;
BEGIN
  TextAttr := 15 + 4 * 16; GotoXy(12, 1);
  Write(Clust:5,' '); GotoXy(24,1); Write(Sect:5,' ');
  GotoXy(68,1); Write(TabFic[0].Attr:2); GotoXy(20,3);
  TextAttr := 4 + 7 * 16; Write('':40); GotoXy(21, 3);
  Write(Chemin); TextAttr := 14+4*16; GotoXy(22,5);
  IF (Sect <= OfsDonnees) THEN
    Write(' R É P E R T O I R E   R A C I N E ')
  ELSE
    Write(' Z O N E   D E S   F I C H I E R S ');
  TextAttr := 15 + 1 * 16; i := 1;
  Max := TabFic[0].Attr;
  WHILE (i <= Max) DO
  BEGIN
    AfficheEntree(TabFic[i], i, (15 + 1 * 16));
    Inc(i);
  END;
  IF (Max < 16) THEN
  BEGIN
    TextAttr := 15 + 1 * 16;
    FOR i := Max + 1 TO 16 DO
    BEGIN
      GotoXy(4, 6 + i); Write('':70);
    END;
  END;
END;
```

➥

Programme Entrees.Pas (suite). ❽

```
PROCEDURE LitClavRep(Clust, Sect : Word);
TYPE Stck   = ARRAY[0..512] OF Word;
VAR Car       : Char;
    NCl, NSc : Word;
    Pile      : Stck;
    i, Max    : Byte;
    EnCours   : Entree;

  FUNCTION Suivant(Sect : Word) : BOOLEAN;
  BEGIN
    LitSectAbs(Drv, Sect + 1, 1, @TabAuxRep);
    Suivant := (TabAuxRep[0] <> 0);
  END;

  FUNCTION Espaces(Chemin : STRING;
                   Nom : Entree) : STRING;
  VAR S1, S2, S3 : STRING;

    FUNCTION VireEspFin(S : STRING) : STRING;
    VAR i : Byte;
    BEGIN
      i := Length(S);
      WHILE ((i >= 1) AND (S[i] = ' ')) DO
      BEGIN
        Delete(S, i, 1);
        Dec(i);
      END;
      VireEspFin := S;
    END;

  BEGIN   { Espaces }
    S1 := VireEspFin(Chemin);
    S2 := VireEspFin(Nom.Nom);
    S3 := VireEspFin(Nom.Ext);
    IF (Chemin[Length(Chemin)] = '\') THEN
      S1 := S1 + S2
    ELSE
      S1 := S1 + '\' + S2;
    IF (S3 > '') THEN
      S1 := S1 + '.' + S3;
    Espaces := S1;
  END; { Espaces }
```

➥

Programme Entrees.Pas (suite). ❾

```
FUNCTION VireRep(S : STRING) : STRING;
BEGIN
  WHILE ((S[Length(S)] IN [#32..#91, #93..#254]) AND
  (Length(S) > 3)) DO
    Delete(S, Length(S), 1);
  IF ((S[Length(S)] = '\') AND (Length(S) > 3)) THEN
    Delete(S, Length(S), 1);
  VireRep := S;
END;

PROCEDURE Push(Val : Word);
BEGIN
  IF (Pile[0] < 512) THEN
  BEGIN
    Inc(Pile[0]);
    Pile[Pile[0]] := Val;
  END
  ELSE
  BEGIN
    GotoXy(12, 24); TextAttr := 4 + 1 * 16;
    Write(' O V E R F L O W : Agrandissez la Pile ');
    ReadLn; ClrScr; Halt;
  END;
END;

FUNCTION Pop : Word;
VAR Val : Word;
BEGIN
  IF (Pile[0] > 0) THEN
  BEGIN
    Val := Pile[Pile[0]];
    Dec(Pile[0]);
    Pop := Val;
  END
  ELSE
    Pop := $FFFF;
END;

PROCEDURE RAZPile;
BEGIN
  Pile[0] := 0;
END;
```

➡

Programme Entrees.Pas (suite). 10

```
BEGIN { LitClavRep }
  Car := #215; NCl := Clust; NSc := Sect; i := 1;
  Max := TabFic[0].Attr; EnCours := TabFic[1]; RAZPile;
  AfficheEntree(EnCours, i, (15 + 4 * 16));
  WHILE (Car <> #27) DO
  BEGIN
    Car := ReadKey;
    CASE Car OF
      #0 : BEGIN
             AfficheEntree(EnCours, i, (15 + 1 * 16));
             Car := ReadKey;
             CASE Car OF
               #80 : BEGIN { Flèche bas }
                       IF (((NSc < OfsDonnees) AND
                         (NCl = 0)) OR ((Sect < SpC)
                         AND (NCl >= 2))) THEN
                       BEGIN
                         IF (i < Max) THEN
                           Inc(i)
                         ELSE
                           IF (Suivant(NSc)) THEN
                           BEGIN
                             Push(NCl); Push(NSc);
                             Inc(NSc); i := Max + 1;
                           END;
                       END
                       ELSE
                         IF (Sect >= SpC) THEN
                         BEGIN
                           IF ((i < Max) AND
                               (Sect = SpC)) THEN
                             Inc(i)
                           ELSE
                           IF ((i >= Max) OR
                               (Sect > SpC)) THEN
                           BEGIN
                             Push(NCl); Push(NSc);
                             NCl := LitFAT(Drv, NCl);
                             IF (NCl > Fin) THEN
                             BEGIN
                               NSc := Pop; NCl := Pop;
                             END
                             ELSE
```

➥

Programme Entrees.Pas (suite). 11

```
                              BEGIN
                                NSc := ((NCl * SpC) +
                               (OfsDonnees)-(SpC SHL 1);
                                i := Max + 1;
                              END;
                            END;
                          END;
                      END;  { Flèche bas }
                #72 : BEGIN { Flèche haut }
                        IF ((NSc < OfsDonnees) AND
                           (NCl = 0)) THEN
                        BEGIN
                           IF ((NSc >= OfsRacine) AND
                              (i > 1)) THEN
                             Dec(i)
                           ELSE
                           IF ((NSc > OfsRacine) AND
                              (i = 1)) THEN
                           BEGIN
                             NSc := Pop; NCl := Pop;
                             i := Max + 1;
                           END;
                        END
                        ELSE
                          IF ((Sect IN [2..SpC]) AND
                             (NCl >= 2)) THEN
                            IF (i > 1) THEN
                              Dec(i)
                            ELSE
                            BEGIN
                              NSc := Pop; NCl := Pop;
                              i := Max + 1;
                            END
                          ELSE
                          IF ((Sect <= 1) AND
                              (NCl > 2)) THEN
                          BEGIN
                            IF (i > 1) THEN
                              Dec(i)
                            ELSE
                            BEGIN
                              NSc := Pop; NCl := Pop;
                              i := Max + 1;
                              IF (NCl = $FFFF) THEN
```

➥

Programme Entrees.Pas (suite). 12

```
                              Halt;
                            Chemin := VireRep(Chemin);
                          END;
                        END
                        ELSE
                        IF ((Sect <= 1) AND
                           (NCl = 2)) THEN
                        BEGIN
                          IF (i > 1) THEN
                            Dec(i)
                          ELSE
                          BEGIN
                            NSc := Pop; NCl := Pop;
                            i := Max + 1;
                            Chemin := VireRep(Chemin);
                          END;
                        END;
                    END; { Flèche haut }
            END; { Case Caractère étendu }
            IF (i = Max + 1) THEN
            BEGIN
              LitRep(Drv, NSc); Max := TabFic[0].Attr;
              i := 1;
              IF (Max > 0) THEN
              BEGIN
                AffRep(NCl, NSc);
                EnCours := TabFic[i];
                AfficheEntree(EnCours, i, (15+4*16));
              END;
            END
            ELSE
            BEGIN
              EnCours := TabFic[i];
              AfficheEntree(EnCours, i, (15+4*16));
            END;
            IF (NSc <= FinRacine) THEN
              Sect := NSc
            ELSE
              Sect := (NSc MOD SpC);
            IF (Sect = 0) THEN
             Sect := 4;
          END;  { Car = #0 }
     #13 :BEGIN
            IF (EnCours.Attr OR $10 = EnCours.Attr)
            THEN
```

Programme Entrees.Pas (suite). 13

```
              BEGIN
                Push(NCl); Push(NSc);
                NCl := EnCours.Clust1;
                NSc := ((NCl * SpC) + OfsDonnees) -
                        (SpC SHL 1);
                i := 1; Sect := 1;
                Chemin := Espaces(Chemin, EnCours);
                LitRep(Drv, NSc); EnCours := TabFic[1];
                Max := TabFic[0].Attr;
                EcranRep; AffRep(NCl, NSc);
                AfficheEntree(EnCours, i, (15+4*16));
              END;
            END; { Car = #13 }
     END;                       { Case }
   END;
 END; { LitClavRep }

BEGIN { Programme principal }
 Sauve := TextAttr;
 IF (ParamCount < 1) THEN
   Drv := LecteurCourant
 ELSE
 BEGIN
   Lect := ParamStr(1); Lect[1] := Upcase(Lect[1]);
   Drv := Ord(Lect[1]) - 65;
 END;
 FillChar(TabRep, SizeOf(TabRep), 0);
 Init(Drv); FinRacine := DernierSectRac(Drv);
 Chemin := Chr(Drv + 65) + ':\'; ChDir(Chemin);
 Secteur := OfsRacine; LitRep(Drv, Secteur); EcranRep;
 AffRep(0, OfsRacine); LitClavRep(0, OfsRacine);
 TextAttr := Sauve; ClrScr;
END.
```

Zone des fichiers : création et effacement de fichiers

La zone des fichiers, c'est ce qui reste après les structures DOS que nous avons décrites dans ce chapitre et le précédent. Autrement dit, ce sont les clusters pointés

par la FAT qui comportent des fichiers, vides ou endommagés. Le Dumper du chapitre 5 (*Disque au niveau logique : le plan d'un disque*) permet la visualisation de cet espace désorganisé. Seules les structures DOS que nous avons présentées dans les deux derniers chapitres permettent de s'y retrouver.

Lorsque le DOS attribue de la place à un nouveau fichier, il fait donc travailler à la fois la FAT, les entrées fichiers et la zone des fichiers. Il commence par écrire l'en-tête correspondant au nouveau fichier. Après quoi, il repère la première entrée libre de la FAT et inscrit son index dans le champ réservé à cet effet. Puis, il continue de chercher autant d'entrées FAT libres qu'il en a besoin et les met à jour en inscrivant le nouvel index trouvé dans l'entrée précédente. La dernière entrée est indiquée par un code compris entre `(F)FF8h` et `(F)FFFh`.

La mise à jour s'effectue en remontant la chaîne des entrées `FAT` à partir de celle pointée par l'entrée fichier jusqu'à la dernière. Une fois celle-ci trouvée, le DOS applique les mêmes principes que lors d'une création, sauf qu'il ne modifie pas le champ «cluster» de l'entrée fichier, mais seulement le champ «taille».

Enfin, un effacement de fichier se poursuit à peu près de la même façon : le DOS remonte la chaîne des clusters appartenant au fichier jusqu'au dernier, et remplit les entrées FAT correspondantes avec la valeur `(0)000h`. Il modifie ensuite le nom du fichier en remplaçant son premier caractère par un «σ» (caractère `229`, ou `E5h`, du code ASCII). Mais il ne change pas le champ de l'entrée fichier qui pointe sur le premier cluster. Grâce à cette particularité, il est possible de récupérer un fichier effacé.

Mais comment fait-on exactement ? Plusieurs opérations sont nécessaires :

1. Lire les entrées fichiers du répertoire concerné jusqu'à en trouver une dont le nom commence par «σ».
2. Mémoriser le numéro de cluster sur lequel elle pointe et calculer le nombre de clusters occupés par le fichier effacé.
3. Lire la FAT au numéro de cluster indiqué par l'entrée fichier concernée et vérifier qu'elle contient bien la valeur `0`. Si ce n'est pas le cas, interrompre le programme.
4. Sinon, lire toutes les entrées FAT, jusqu'à ce que l'on en ait trouvé le nombre nécessaire, qui contiennent la valeur zéro et mémoriser leurs emplacements.
5. S'il n'y a pas eu d'erreur, demander à l'utilisateur d'entrer le premier caractère du nom de fichier, le mémoriser.
6. Pour chaque entrée FAT mémorisée, y inscrire l'index de la suivante : la première entrée contient l'index de la seconde, et ainsi de suite jusqu'à la dernière que l'on remplit avec la valeur `FFFFh`.
7. Ecrire la FAT sur le disque, ainsi que l'entrée fichier dont on a modifié le nom.

8. Afficher un message indiquant que tout s'est bien passé, et en demandant à l'utilisateur de vérifier l'intégralité du fichier.

Cela n'a rien de très compliqué, mais demande tout de même de prendre des précautions : le programme `UnDel.Pas` fonctionne uniquement sur disque dur et ne récupère qu'un seul fichier à la fois. Pour bien faire, il aurait fallu enregistrer les numéros de clusters de tous les fichiers effacés dans une liste, de telle manière que l'on soit sûr de ne pas réécrire sur un emplacement appartenant à un autre fichier. Nous laissons cette modification en exercice au lecteur : tel qu'il est, le programme fait déjà 350 lignes.

Procédure	***Type***	***Description***	***Ligne***
`Init`	P	Lit le secteur boot et initialise des variables globales	*31*
`Push`	P	Dépose une valeur au sommet de la pile passée en paramètre	*45*
`Pop`	F	Renvoie la valeur du sommet de la pile passée en paramètre	*59*
`RAZ`	P	Initialise la pile passée en paramètre	*75*
`LitFAT`	F	Renvoie la valeur contenue dans l'entrée FAT dont on a passé le numéro	*80*
`ChercheEfface`	F	Cherche une entrée fichier effacée dans le répertoire, demande si l'on souhaite la restaurer et le fait. Renvoie une valeur booléenne	*90*
`\FATLibre`	P	Cherche les entrées FAT de valeur zéro et les empile	*97*
`\ModifieFAT`	P	Dépile les entrées FAT et modifie la FAT (récursive)	*119*
`ChercheRep`	F	Renvoie le numéro de cluster du répertoire où se trouve le fichier effacé	*209*
`\Caps`	F	Renvoie une chaîne en majuscules	*214*
`\DecoupeChemin`	F	Renvoie le nom de chemin relatif et la suite éventuelle du nom de chemin	*227*
`\Egalite`	F	Renvoie la valeur `TRUE` si les deux chaînes passées en paramètres sont égales	*249*
`Main`	P	Procédure principale du programme (récursive)	*320*
`Programme`			*338*

Dépendances :

Procédure	Appelle procédure
Init	LitSectAbs
Push	
Pop	
RAZ	
LitFAT	LitSectAbs
ChercheEfface	LitSectAbs
	EcritSectAbs
	RAZ
	LitFAT
\FATLibre	Push
	Pop
	RAZ
	LitFAT
\ModifieFAT	LitSectAbs
	EcritSectAbs
	Pop
	ChercheEFface\ModifieFAT
ChercheRep	LitSectAbs
	LitFAT
	ChercheRep\Caps
	ChercheRep\DecoupeChemin
	ChercheRep\Egalite
\Caps	
\DecoupeChemin	
\Egalite	
Main	LitFAT
	ChercheEfface
	Main
Programme	Init
	ChercheRep
	Main

Tableau 7.10
Références croisées de UnDel.Pas.

Listing 7.11
Programme UnDel.Pas. ❶

```
PROGRAM RecupereFichier;          { UnDel.Pas }

USES Dos, Crt;

CONST MaxItem = 512;

TYPE OctPtr    = ^Byte;
     Tableau   = ARRAY[0..511] OF Byte;
     Stck      = ARRAY[0..MaxItem] OF Word;
     EntreeFic = RECORD
                  Nom    : String[12];
                  Attr   : Byte;
                  Taille : LongInt;
                  Clust1 : Word;
                 END;

VAR TabFAT, TabFic  : Tableau;
    OpS, SpF, Clust, OfsDonnees,
    OfsRacine       : Word;
    SpC, NbF, Drv   : Byte;
    Entree          : EntreeFic;
    Parametres      : String;

 {$L D:\Pascal\Sources\AbsRead.Obj}
 {$F+}
 PROCEDURE LitSectAbs(Drv : Byte; No, Nb : Word;
                      Tab : OctPtr); EXTERNAL;
 PROCEDURE EcritSectAbs(Drv : Byte; No, Nb : Word;
                        Tab : OctPtr); EXTERNAL;
 {$F-}

 PROCEDURE Init(Drv : Byte);
 VAR NbE : Word;
 BEGIN
   LitSectAbs(Drv, 0, 1, @TabFic);
   OpS := (TabFic[$C] SHL 8) + TabFic[$B];
   SpC := TabFic[$D];
   NbF := TabFic[$10];
   SpF := (TabFic[$17] SHL 8) + TabFic[$16];
   NbE := (TabFic[$12] SHL 8) + TabFic[$11];
   OfsRacine := (NbF * SpF) + ((TabFic[$F] SHL 8) +
                               TabFic[$E]);
   OfsDonnees := OfsRacine + ((NbE SHL 5) DIV OpS);
   FillChar(TabFic[0], SizeOf(TabFic), 1);
 END;
```

➥

Programme UnDel.Pas (suite). ❷

```
PROCEDURE Push(Val : Word; VAR Pile : Stck);
BEGIN
  IF (Pile[0] >= MaxItem) THEN
  BEGIN
    Write(' OVERFLOW dans Pile ');
    Halt;
  END
  ELSE
  BEGIN
    Inc(Pile[0]);
    Pile[Pile[0]] := Val;
  END;
END;

FUNCTION Pop(VAR Pile : Stck) : Word;
VAR Res : Word;
BEGIN
  IF (Pile[0] = 0) THEN
  BEGIN
    Write(' UNDERFLOW dans Pile ');
    Halt;
  END
  ELSE
  BEGIN
    Res := Pile[Pile[0]];
    Dec(Pile[0]);
    Pop := Res;
  END;
END;

PROCEDURE RAZ(VAR Pile : Stck);
BEGIN
  Pile[0] := 0;
END;

FUNCTION LitFAT(Drv : Byte; NoC : Word) : Word;
VAR Sect, Idx, Res : Word;
BEGIN
  Sect := (NoC DIV (OpS SHR 1)) + 1;
  LitSectAbs(Drv, Sect, 1, @TabFAT);
  Idx := (NoC MOD (OpS SHR 1)) SHL 1;
  Res := (TabFAT[Idx + 1] SHL 8) + TabFAT[Idx];
  LitFAT := Res;
END;
```

➥

Programme UnDel.Pas (suite). ❸

```
FUNCTION ChercheEfface(Drv : Byte; No : Word) : Boolean;
VAR i, NbEntFAT : Word;
    Recup, Ok   : Boolean;
    Reponse     : Char;
    NbO         : LongInt;
    Pile1       : Stck;

  PROCEDURE FATLibre(Drv : Byte; Premier, Nb : Word);
  VAR Index, Valeur, Fait : Word;
      Pile2               : Stck;
  BEGIN
    Index := Premier + 1; RAZ(Pile2); Fait := 0;
    REPEAT
      Valeur := LitFAT(Drv, Index);
      IF (Valeur = 0) THEN
      BEGIN
        Push(Index, Pile2);
        Inc(Fait);
      END;
      Inc(Index);
    UNTIL (Fait + 1 = Nb);
    Push($FFFF, Pile2); Inc(Fait);
    REPEAT
      Valeur := Pop(Pile2);
      Push(Valeur, Pile1);
      Dec(Fait);
    UNTIL (Fait = 0);
  END;

  PROCEDURE ModifieFAT(Drv : Byte; Premier, Nb : Word);
  VAR Cluster, Suivant, i        : Word;
      LimiteSect, Index, SectFAT : Word;
  BEGIN
    Cluster := Premier; i := 1;
    SectFAT := (Premier DIV (OpS SHR 1)) + 1;
    LimiteSect := SectFAT * (OpS SHR 1);
    WHILE ((Cluster <= LimiteSect) AND (i <= Nb)) DO
    BEGIN
      Suivant := Pop(Pile1);
      Index := (Cluster MOD (OpS SHR 1) SHL 1);
      TabFAT[Index + 1] := Hi(Suivant);
      TabFAT[Index] := Lo(Suivant);
      Cluster := Suivant; Inc(i);
    END;
```

➥

Programme UnDel.Pas (suite). ❹

```
      EcritSectAbs(Drv, SectFAT, 1, @TabFAT);
      EcritSectAbs(Drv, SpF + SectFAT, 1, @TabFAT);
      IF ((Cluster >= LimiteSect) AND (i < Nb)) THEN
      BEGIN
        SectFAT := (Cluster DIV (OpS SHR 1)) + 1;
        LitSectAbs(Drv, SectFAT, 1, @TabFAT);
        ModifieFAT(Drv, Cluster, (Nb - i));
      END;
    END;

  BEGIN
    LitSectAbs(Drv, No, 1, @TabFic);
    i := 0; Ok := False; RAZ(Pile1); Reponse := 'Â';
    WHILE (i < (512 - 32)) DO
    BEGIN
      IF (TabFic[i] <> $E5) THEN  { 'Â' = #229 = #$E5 }
        Inc(i, 32)
      ELSE
      BEGIN
        Ok := True;
        WITH Entree DO
        BEGIN
          Nom[0] := #11;
          Move(Mem[Seg(TabFic):Ofs(TabFic)+i],
               Nom[1], 11);
          Attr := TabFic[i + $B];
          Taille:=TabFic[i+$1F]; Taille:=Taille SHL 24;
          NbO := TabFic[i+$1E] SHL 16; Inc(Taille, NbO);
          NbO := TabFic[i+$1D] SHL 8; Inc(Taille, NbO);
          Inc(Taille, TabFic[i + $1C]);
          Clust1:=(TabFic[i+$1B] SHL 8)+TabFic[i+$1A];
          Recup := (LitFAT(Drv, Clust1) = 0);
          IF (NOT Recup) THEN
          BEGIN
            WriteLn('Fichier ', Nom, ' irrécupérable ');
            Inc(i, 32);
          END
          ELSE
          BEGIN
            NbEntFAT := Taille DIV (SpC * OpS);
            IF (Taille MOD (SpC * OpS) <> 0) THEN
              Inc(NbEntFAT);
            FATLibre(Drv, Clust1, NbEntFAT);
            Write('Le fichier ',Nom,' peut ');
```

➥

Programme UnDel.Pas (suite). ❺

```
            WriteLn('être récupéré');
            Write('sa taille est de ', Taille, ' octets,');
            Write(' et il occupe ',NbEntFAT);
            WriteLn(' clusters');
            Write('Souhaitez-vous le récupérer ');
            Write('(O/N) ? : ');
            WHILE NOT(Reponse IN ['O', 'N']) DO
            BEGIN
              Reponse := ReadKey;
              Reponse := Upcase(Reponse);
            END;
            WriteLn(Reponse);
            IF (Reponse = 'O') THEN
            BEGIN
              WriteLn; Reponse := 'Â';
              Write('Donnez la première lettre : ');
              WHILE NOT(Reponse IN ['0'..'9', 'A'..'Z']) DO
              BEGIN
                Reponse := ReadKey;
                Reponse := Upcase(Reponse);
              END;
              WriteLn(Reponse);
              Nom[1] := Reponse; TabFic[i] := Ord(Nom[1]);
              ModifieFAT(Drv, Clust1, NbEntFAT);
              EcritSectAbs(Drv, No, 1, @TabFic);
              Write('Fichier ', Nom, ' récupéré : ');
              WriteLn('vérifiez son intégralité ');
            END;
          END;
        END;
      END;
    END;
    ChercheEfface := Ok;
  END;

  FUNCTION ChercheRep(Drv : Byte; Cluster : Word;
                      S : String) : Word;
  VAR Chaine, Nom          : String;
      Secteur, SectMax, i  : Word;
      Trouve               : Boolean;

    FUNCTION Caps(S : String) : String;
    VAR Chaine : String;
        i      : Byte;
```

➥

Programme UnDel.Pas (suite). ❻

```
    BEGIN
      i := 1; Chaine := '';
      WHILE (i <= Length(S)) DO
      BEGIN
        Chaine := Chaine + Upcase(S[i]);
        Inc(i);
      END;
      Caps := Chaine;
    END;

    FUNCTION DecoupeChemin(VAR S1 : String) : String;
    VAR Chaine : String;
    BEGIN
      IF (Pos('\', S1) <> 0) THEN
      BEGIN
        Delete(S1, 1, Pos('\', S1));
        IF (Pos('\', S1) <> 0) THEN
        BEGIN
          Chaine := Copy(S1, 1, Pos('\', S1) - 1);
          S1 := Copy(S1, Pos('\', S1), Length(S1));
        END
        ELSE
        BEGIN
          Chaine := S1;
          S1 := '';
        END;
      END
      ELSE
        Chaine := '';
      DecoupeChemin := Chaine;
    END;

    FUNCTION Egalite(S1, S2 : String) : Boolean;
    VAR i  : Byte;
        Ok : Boolean;
    BEGIN
      i := Length(S1); Ok := True;
      IF (i = Length(S2)) THEN
      BEGIN
        WHILE ((Ok) AND (i > 0)) DO
        IF (S1[i] <> S2[i]) THEN
          Ok := False
        ELSE
          Dec(i);
      END
```

➡

Programme UnDel.Pas (suite). ❼

```
      ELSE
        Ok := False;
      Egalite := Ok;
    END;

  BEGIN
    S := Caps(S); Chaine := S; i := 0;
    Chaine := DecoupeChemin(S); Trouve := False;
    Secteur := OfsRacine; SectMax := OfsDonnees - 1;
    LitSectAbs(Drv, Secteur, 1, @TabFic);
    WHILE ((TabFic[i] <> 0) AND NOT(Trouve)) DO
    IF (i >= (512 - 32)) THEN
    BEGIN
      i := 0; Inc(Secteur);
      IF (Secteur <= SectMax) THEN
        LitSectAbs(Drv, Secteur, 1, @TabFic)
      ELSE
      IF (Cluster >= 2) THEN
      BEGIN
        Cluster := LitFAT(Drv, Cluster);
        Secteur:=((Cluster*SpC)+OfsDonnees)-(SpC SHL 1);
        SectMax := Secteur + SpC; Nom := ''; i := 0;
      END
      ELSE
      BEGIN
        WriteLn(' Erreur dans le nom de chemin : ', S);
        Halt(1);
      END;
    END
    ELSE
    BEGIN
      Cluster := TabFic[i + $1B] SHL 8 + TabFic[i + $1A];
      Nom[0] := #11;
      Move(TabFic[i], Nom[1], 11);
      IF (Pos('.', Chaine) <> 0) THEN
        Nom := Copy(Nom, 1, 8) + '.' + Copy(Nom, 9, 11);
      WHILE (Pos(' ', Nom) <> 0) DO
        Delete(Nom, Pos(' ', Nom), 1);
      IF (Egalite(Chaine, Nom)) THEN
      BEGIN
        Chaine := DecoupeChemin(S);
        IF (S = '') THEN
          Trouve := True
        ELSE
```

➥

Programme UnDel.Pas (suite). ❽

```
        BEGIN
          WriteLn('Répertoire ', Nom);
          Secteur:=((Cluster*SpC)+OfsDonnees) -
                   (SpC SHL 1);
          SectMax := Secteur + SpC; Nom := ''; i := 0;
          LitSectAbs(Drv, Secteur, 1, @TabFic);
        END;
      END
      ELSE
        Inc(i, 32);
    END;
    WriteLn('Répertoire ', Nom);
    ChercheRep := Cluster;
  END;

  PROCEDURE Main(Drv : Byte; Clust : Word);
  VAR Sect, i : Word;
      Ok      : Boolean;
  BEGIN
    Sect := ((Clust * SpC) + OfsDonnees) - (SpC SHL 1);
    Ok := False; i := Sect;
    WHILE ((i <= Sect + (SpC - 1)) AND NOT(Ok)) DO
    BEGIN
      Ok := ChercheEfface(Drv, i);
      Inc(i);
    END;
    IF NOT Ok THEN
    BEGIN
      Clust := LitFAT(Drv, Clust);
      Main(Drv, Clust);
    END;
  END;

BEGIN
  IF (ParamCount < 1) THEN
  BEGIN
    Write('Paramètres : lecteur, chemin ');
    WriteLn('nom de fichier');
    Halt;
  END
```

➥

Programme UnDel.Pas (suite). ❾

```
  ELSE
  BEGIN
    Parametres := ParamStr(1);
    Parametres[1] := Upcase(Parametres[1]);
    Drv := Ord(Parametres[1]) - 65;
    Init(Drv);
    Clust := ChercheRep(Drv, 0, Parametres);
    Main(Drv, Clust);
  END;
END.
```

Attention ! Ce programme n'effectuant aucune sauvegarde de la FAT et ne vérifiant pas que l'entrée FAT qu'il lit appartient à un autre fichier effacé, il ne peut fonctionner de manière sûre que dans un répertoire où un seul fichier est effacé (vérifiez avec `Entree.Pas`). Dans ces conditions d'utilisation, il est parfaitement au point. Nous conseillons toutefois aux utilisateurs désirant le tester d'effectuer auparavant une copie de leur FAT (programmes `Mirror`, ou `UnFormat`).

Conclusion

Il n'y a plus grand chose à dire – ou à écrire – sur les disques et la façon dont le DOS les gère. On pourrait toutefois imaginer énormément d'autres programmes : de la carte d'occupation du disque jusqu'aux copies d'un disque à l'autre (`DiskCopy`) en passant par les comparaisons et les sauvegardes de FAT, tout est imaginable et relativement simple à réaliser. L'écriture de tels utilitaires nécessite seulement du temps, une bonne analyse préalable des tests à effectuer, et une sacrée dose de patience. Nous ne pouvons qu'encourager les lecteurs à réaliser eux-mêmes ce genre d'utilitaires.

Chapitre 8

Fichiers de données

Mots-clefs

Disk Transfer Area	La DTA (zone de transfert disque) est une structure chargée de mémoriser les entrées fichiers en provenance du disque. Elle se trouve dans les 128 derniers octets du PSP. On doit lui donner une nouvelle adresse grâce à la fonction 1Ah de l'Int 21h avant de l'utiliser.
File Handle Table	La *File Handle Table* (FHT ou table des handles) est une structure qui mémorise les handles utilisés par un programme. Elle se trouve à l'offset 18h du PSP et chacun de ses enregistrements pointe sur une entrée de la *System File Table,* qui contient le descriptif complet du fichier pointé.
Filtres	Les filtres sont des programmes qui utilisent les possibilités de redirections offertes par le DOS. Ils peuvent ainsi envoyer les résultats de leurs traitements aussi bien sur l'écran, sur l'imprimante, ou sur un fichier.
Handles	C'est un numéro d'identification de fichier que le DOS renvoie au programme ayant ouvert le fichier. Cette technique de désignation des fichiers facilite les opérations du DOS et du programmeur. 15 handles sont disponibles par programme, plus cinq qui sont prédéfinis et correspondent aux périphériques standard d'entrées/sorties.
Piping	Technique qui consiste pour un programme à accepter en entrée les sorties d'un programme précédent. Sa mise en œuvre est rendue aisée grâce aux fonctions handles du DOS ainsi qu'à ses possibilités de redirection.
Redirection	La redirection est une des fonctionnalités du DOS qui permet d'employer un fichier ou un périphérique en lieu et place d'un autre fichier ou périphérique. Ainsi peut-on rediriger les sorties d'un programme sur l'imprimante au lieu de l'écran (ou l'inverse).
System File Table	La SFT (table des fichiers du système) mémorise la description complète de chaque fichier ou périphérique ouvert et pointé par un handle. Le DOS dispose de deux FHT : l'une fonctionne avec les handles, l'autre avec les FCB. On trouve son adresse par l'intermédiaire de la fonction 52h de l'Int 21h.

DOS signifie système d'exploitation de disque : on ne s'étonnera donc pas qu'il offre de nombreuses possibilités de gestion des fichiers et des répertoires. Dans les prochains chapitres, nous ne nous intéresserons cependant qu'aux fichiers : la notion de fichier recouvre en effet aussi bien celle de données (ce que l'on manipule) que celle de programme (ce qui manipule) et – naturellement – celle de mémoire. Un répertoire est une structure d'organisation du disque. Un fichier mémorise de l'information, qu'elle soit statique (données) ou dynamique (programme).

Dans ce chapitre, nous nous intéressons aux fichiers de données et à leur gestion par le DOS. Le système d'exploitation gère pourtant de la même façon tous les fichiers, mais leurs différences de nature font que certains (comme les programmes) nécessitent des traitements supplémentaires. Ces traitements et la façon dont les fichiers en question sont stockés sur le disque (leur format) feront l'objet de notre prochain chapitre.

Les principaux points que nous abordons ici sont :

1. les principes de la gestion par handle et les fonctions handles du DOS ;
2. la gestion interne des handles par le DOS ;
3. la redirection des données et les filtres.

Il nous est apparu inutile d'encombrer ce livre de passages définitivement obsolètes : les FCB (File Control Blocks) sont un héritage de CP/M et ne servent pratiquement plus. Nous ne les aborderons donc pas.

Gestion de fichiers par handles

Contrairement au Pascal, de nombreux autres langages comme le BASIC, le Modula 2 ou le C, renvoient un numéro lorsqu'on leur demande d'effectuer une opération d'ouverture ou de création de fichier. Ce qui peut parfois dérouter les débutants en programmation n'est en fait que la concrétisation du mode d'accès DOS aux fichiers. Le numéro est le handle du fichier. Une fois qu'on le connaît, il n'est plus nécessaire de mémoriser le chemin et le nom de fichier pour effectuer une entrée/sortie : le handle identifie le fichier. Cela permet au DOS de gagner du temps, et donc aux opérations de se dérouler plus vite. En outre, on peut rediriger les sorties d'un programme sans que celui-ci le sache. Les périphériques d'entrées/sorties (clavier, écran, imprimante, port série) sont alors considérés comme des fichiers. Enfin, contrairement aux FCB, les handles supportent l'organisation logique du disque (chemins d'accès absolus et relatifs).

Malheureusement, le nombre de handles disponibles est limité. Il est au maximum de 255 (version 3.0 du DOS) pour l'ensemble du système et de 15 par programme.

Cinq handles, utilisables en plus par tous les programmes, sont affectés aux périphériques suivants :

Numéro de handle	Correspondance et périphérique utilisé
0	Entrée standard (clavier, périphérique CON)
1	Sortie standard (écran, périphérique CON)
2	Erreur standard (écran, périphérique CON)
3	Auxiliaire (périphérique AUX, port COM1)
4	Parallèle (périphérique PRN, port LPT1)

Tableau 8.1
Handles prédéfinis.

Le nombre de handles maximum correspond à la directive «FILES = nnn» inclue dans le fichier CONFIG.SYS. Si le programmeur cherche à obtenir un numéro d'identification de fichier excédant la limite des 20 (au total) auquel son programme a droit, le code d'erreur numéro 4 lui sera retourné par le DOS. Enfin, avant d'aborder les fonctions handles du DOS, il faut signaler que de nombreuses opérations sur les fichiers peuvent entraîner un appel à l'Int 24h (interruption d'erreur critique). Tout programme faisant intensivement appel aux fichiers devrait donc détourner ce vecteur d'interruption, de façon à pouvoir en contrôler les effets.

Numéro d'erreur	Description	Version DOS
2	Fichier introuvable	2.x
3	Chemin d'accès introuvable	2.x
4	Trop de fichiers ouverts	2.x
5	Accès refusé (fichier en lecture seule)	2.x
6	Handle inconnu	2.x
12h	Fichier perdu	2.x
1Dh	Impossible d'écrire (code étendu)	3.x
1Eh	Impossible de lire (code étendu)	3.x
48h	Redirection disque/imprimante interrompue (code étendu)	3.x
50h	Fichier existant	3.x
54h	Trop de redirections	3.x
55h	Redirection déjà en cours	3.x

Tableau 8.2
Codes d'erreurs DOS pour les fonctions handles.

Les codes d'erreurs étendus s'obtiennent en faisant appel à la fonction 59h de l'Int 21h. DOS renvoie alors le type de l'erreur, le type d'action à effectuer et l'emplacement de l'erreur. Un grand programme doit prendre ces renseignements en compte pour être certain de bien fonctionner.

Fonctions handle du DOS

Le tableau qui suit indique quelles sont les fonctions handles que l'on peut appeler par l'intermédiaire de l'interruption 21h du DOS. Il tient seulement compte des fonctions concernant les fichiers.

Nom de la fonction	*Appel*	*Retour*
Création de fichier	AH := 3Ch CX := Attribut DS := Seg(Chemin) DX := Ofs(Chemin)	AX = handle Si CF = 1, AX = Erreur
Ouverture de fichier	AH := 3Dh AL := Mode d'accès DS := Seg(Chemin) DX := Ofs(Chemin)	AX = handle Si CF = 1, AX = Erreur
Fermeture de fichier	AH := 3Eh BX := handle	Si CF = 1, AX = Erreur
Lecture fichier ou périphérique	AH := 3Fh BX := handle CX := nombre d'octets à lire DS := Seg(Buffer) DX := Ofs(Buffer)	AX = nombre d'octets lus Si CF = 1, AX = Erreur
Ecriture fichier ou périphérique	AH := 40h BX := handle CX := nombre d'octets à écrire DS := Seg(Buffer) DX := Ofs(Buffer)	AX = octets écrits Si CF = 1, AX = Erreur
Effacement fichier	AH := 41h DS := Seg(Chemin) DX := Ofs(Chemin)	Si CF = 1, AX = Erreur
Déplacer pointeur de fichier (Seek)	AH := 42h AL := Méthode BX := handle CX := Hi(position) DX := Lo(position)	DX = Hi(position) AX = Lo(position) Si CF = 1, AX = Erreur

➥

(suite du tableau)

Nom de la fonction	Appel	Retour
Accès aux attributs	AH := 43h AL := Mode Si Mode = écriture CX := Attribut DS := Seg(Chemin) DX := Ofs(Chemin)	CX = Attribut Si CF = 1, AX = Erreur
Duplique handle	AH := 45h BX := handle1	AX = handle numéro 2 Si CF = 1, AX = Erreur
Redirige handle	AH := 46h BX := handle1 CX := handle2	Si CF = 1, AX = Erreur
Trouve premier fichier	AH := 4Eh CX := Attribut DS := Seg(Chemin) DX := Ofs(Chemin)	DTA mise à jour Si CF = 1, AX = Erreur
Trouve fichiers suivants	AH := 4Fh	Si CF = 1, AX = Erreur
Renomme fichier	AH := 56h DS := Seg(Chemin1) DX := Ofs(Chemin1) ES := Seg(Chemin2) DI := Ofs(Chemin2)	Si CF = 1, AX = Erreur
Accès date et heure	AH := 57h AL := Mode BX := handle Si Mode = écriture CX := Heure DX := Date	Si Mode = lecture, CX = Heure, DX = Date Si CF = 1, AX = Erreur
Création fichier temporaire	AH := 5Ah CX := Attribut DS := Seg(Chemin) DX := Ofs(Chemin)	AX = handle DS = Seg(Chemin) DX = Ofs(Chemin) Si CF = 0, AX = Erreur
Création nouveau fichier	AH := 5Bh CX := Attribut DS := Seg(Chemin) DX := Ofs(Chemin)	AX = handle Si CF = 1, AX = Erreur
Fixe maximum handles	AH := 67h BX := Nombre de handles	Si CF = 1, AX = Erreur
Vide buffer	AH := 68h BX := handle	Si CF = 1, AX = Erreur

Tableau 8.3
Les fonctions handle fichiers de l'Int 21h.

On aura remarqué une assez grande homogénéité dans les codes de retour du DOS : soit CF est à un et AX renvoie le code d'erreur, soit CF est vide et – s'il y a lieu – AX renvoie le numéro d'identification du fichier. En entrée, le registre AH contient *toujours* le numéro de la fonction appelée.

Les attributs de fichier sont codés sur 8 bits et se présentent de la façon indiquée au chapitre 7 (*Disque au niveau logique : structures DOS de haut niveau V ; tableau 7.3*). Le chemin est une chaîne terminée par un caractère nul de code ASCII 0 (chaîne ASCIIZ), et indique la plupart du temps le nom de chemin (relatif ou absolu) et le nom de fichier. Seule la fonction de création d'un fichier temporaire fait exception à cette règle : la chaîne doit alors contenir uniquement le nom de chemin, suivi de 13 caractères nuls consécutifs. La fonction utilise ces 13 zéros lorsqu'elle crée le nom du fichier. C'est aussi pour cette raison qu'elle renvoie ensuite la chaîne modifiée à l'appelant. Il y a deux fonctions de création de fichier : la seconde (fonction 5Bh) échoue si le fichier existe déjà, tandis que la première (fonction 3Ch) écrase le fichier existant, à moins qu'il ne soit en lecture seule, auquel cas elle échoue. Il existe également une troisième fonction de création / ouverture / déplacement de fichier sous DOS 4.0 (fonction 6Ch).

Fonction 3Dh : ouverture de fichier

Le mode d'accès est un octet codé de la façon suivante :

Bit	*Signification*
0-2	Mode d'accès : 000 si lecture seule 001 si écriture seule 010 si lecture/écriture
3	Réservé
4-6	Mode de partage en réseau : 000 compatible 001 aucun accès étranger 010 pas d'accès en écriture 011 pas d'accès en lecture 100 accepter tous les accès
7	Drapeau d'héritage : 0 Le handle peut être transmis au processus fils 1 Le handle est réservé au père

Tableau 8.4
Octet de mode d'accès pour la fonction 3Dh.

Cette fonction permet d'ouvrir des fichiers possédant n'importe quel attribut. Si le fichier est en lecture seule, il faut cependant que le mode d'accès soit à 001.

Fonction 3Eh : fermeture de fichier

Si l'on demande la fermeture du fichier possédant le handle zéro, le clavier ne répondra plus.

Fonction 3Fh : lecture de fichier ou périphérique

Cette fonction, comme celle d'écriture qui lui correspond, a deux intérêts : elle permet des accès bufferisés et elle facilite extrêmement la redirection. Elle correspond à la procédure `BlockRead` du Turbo Pascal, mais va beaucoup plus loin : il est possible de lui passer un tampon de 64 Ko, elle renvoie le nombre exact d'octets lus et permet de lire le clavier.

Si cette fonction renvoie `AX` et `CF` à zéro, le pointeur de fichier était déjà en dernière position au moment de la lecture. Si le nombre d'octets lus est inférieur au nombre d'octets dont on a demandé la lecture, le pointeur de fichier est arrivé en fin de course (cas des «restes», lorsqu'on lit un fichier secteur par secteur). Si l'on a demandé la lecture du clavier et que celui-ci est en mode ASCII (*cooked*), l'opération s'arrêtera d'elle-même au premier retour charriot.

Fonction 40h : écriture sur fichier ou périphérique

C'est le pendant de la précédente, dont elle possède les qualités. Ses limitations sont en revanche différentes.

Si l'on appelle cette fonction en lui demandant d'écrire `0` octet et que le pointeur de fichier se trouve en première position, elle vide le fichier. Si le pointeur se trouve en dernière position, elle l'étend. Si la fonction retourne un nombre d'octets écrits inférieur à celui demandé et que l'on écrit sur un fichier, il n'y a probablement plus assez de place sur le disque. Si l'on écrit sur un dispositif caractère (écran, imprimante, etc.), il y a sans doute un code `^Z (1Ah, 26)` dans les données du buffer.

Fonction 41h : effacement de fichier

Cette fonction ne permet pas d'effacer plusieurs fichiers à la fois : la chaîne ASCIIZ transmise ne doit donc pas contenir de caractères jokers. Il n'est pas non plus possible d'effacer un fichier en lecture seule avec cette fonction.

Fonction 42h : déplacement du pointeur de fichier

L'octet de méthode peut prendre 3 valeurs :

Méthode choisie	*Signification*
`00h`	Déplacement absolu (à partir du début du fichier)
`01h`	Déplacement relatif (à partir de la position actuelle du pointeur dans le fichier)
`02h`	Déplacement absolu à partir de la fin du fichier

Tableau 8.5
Valeurs de l'octet de méthode de la fonction 42h.

On trouve la taille d'un fichier en déplaçant le pointeur de 0 octet à partir de la fin du fichier. Les méthodes 01h et 02h peuvent déplacer le pointeur hors du fichier : aucune erreur ne sera détectée à ce stade.

Fonction 43h : accès à l'attribut du fichier

L'octet de méthode d'accès peut prendre deux valeurs :

Méthode choisie	*Signification*
00h	Lecture de l'attribut de fichier
01h	Ecriture de l'attribut de fichier

Tableau 8.6
Valeurs de l'octet de méthode de la fonction 43h.

Il est évidemment impossible de positionner le bit de volume ou le bit de répertoire d'un fichier.

Fonction 45h : duplique un handle

Cette fonction permet de disposer d'un nouveau numéro d'accès au fichier. Elle est surtout utile dans le cas de la redirection de fichiers, ou lorsqu'on souhaite mettre à jour l'entrée du répertoire correspondant au fichier sans pour autant perdre la possibilité d'y accéder. Dans ce cas, on duplique le handle original, on ferme le handle dupliqué et l'on continue de travailler avec l'original. Si l'on fermait par inadvertance le handle original (et non le handle dupliqué), le handle dupliqué serait également fermé. En effet, toutes les opérations affectant l'original sont reportées sur la copie, tandis que l'inverse n'est pas vrai.

Fonction 46h : redirige un handle

Cette fonction copie les caractéristiques du fichier pointé par le handle original dans celles du fichier pointé par la copie : la copie est alors redirigée. En effet, si l'on copie le descripteur de périphérique se rapportant à l'imprimante vers le descripteur de périphérique écran, toutes les opérations d'impression seraient reportées vers l'écran. Il s'agit donc là d'une fonction particulièrement puissante du DOS, notamment lors de l'exécution de programmes fils.

Si le handle à rediriger pointait un fichier ouvert au moment de l'appel à cette fonction, le fichier serait d'abord fermé avant d'être redirigé.

Fonction 4Eh : recherche du premier fichier

L'appel à cette fonction suppose que l'on ait d'abord demandé à affecter la DTA à un buffer (fonction `1Ah`). Le résultat de la fonction est envoyé dans la DTA sous la forme suivante, qui est aussi celle du type `SearchRec` en Turbo Pascal (unité DOS) :

Description de champ	***Offset***
Réservé au DOS	`00h-14h`
Attribut du fichier	`15h`
Heure de dernier accès	`16h-17h`
Date de dernier accès	`18h-19h`
Taille du fichier	`1Ah-1Dh`
Nom, point et extension	`1Eh-2Ah`

Tableau 8.7
Format de la DTA en sortie des fonctions 4Eh et 4Fh.

La fonction d'affectation de la DTA (fonction `1Ah`) a besoin des paramètres suivants :

Appel	***Retour***
`AH := 1Ah`	Rien
`DS := Seg(NouveauBuffer)`	
`DX := Ofs(NouveauBuffer)`	

Tableau 8.8
Appel de la fonction 1Ah.

La fonction `4Eh` accepte l'emploi de caractères jokers dans la chaîne ASCIIZ de spécifications.

Fonction 4Fh : recherche du fichier suivant

On appelle cette fonction à la suite de la précédente : les résultats sont renvoyés dans la même zone DTA.

Fonction 56h : renommer un fichier

Cette fonction permet également de changer un fichier de répertoire (mais pas de disque logique). Elle n'accepte pas l'emploi de caractères jokers.

Fonction 57h : accès à la date et à l'heure

L'octet de méthode d'accès peut prendre deux valeurs :

Méthode choisie	*Signification*
`00h`	Lecture de la date et de l'heure
`01h`	Ecriture de la date et de l'heure

Tableau 8.9
Valeurs de l'octet de méthode de la fonction 57h.

La date et l'heure sont codées de la même façon que nous l'avions expliqué au chapitre 7 (*Disques au niveau logique : structures DOS de haut niveau*). Le fichier doit être ouvert pour que cette fonction donne un résultat correct (en revanche, il est possible de modifier la date et l'heure directement sur le disque sans ouvrir le fichier). Si la date et l'heure sont fixées à zéro, elles n'apparaîtront pas lors d'une commande `DIR`.

Fonction 5Ah : création d'un fichier temporaire

Cette fonction crée elle-même le nom du fichier temporaire : la chaîne ASCIIZ qu'on lui passe en paramètre doit donc contenir au moins 13 caractères nuls. Les 12 premiers serviront à la création du nom, et le dernier terminera la chaîne. Les fichiers temporaires ne sont pas automatiquement effacés lors de la fin du programme.

Fonction 5Bh : création d'un nouveau fichier

Cette fonction est similaire à la fonction `3Ch`, à la différence qu'elle échouera si un fichier de même nom existe déjà dans le répertoire (la fonction `3Ch` écraserait le fichier).

Fonction 67h : fixe le compteur de handles

Cette fonction permet d'élargir la taille de la table réservée aux handles. Toutefois, s'il n'y a pas assez de place pour affecter un nouveau bloc aux handles et si la demande est supérieure à 20, la fonction retourne un code d'erreur. Aucune erreur n'est cependant détectée si le nombre de handles demandés est supérieur à celui indiqué par la directive «`FILES=nnn`» du fichier `CONFIG.SYS`.

Fonction 68h : vide le buffer d'un fichier

Cette fonction permet d'écrire les buffers d'un fichier dans ce fichier et de mettre à jour la date, l'heure et la taille du fichier sans avoir à le fermer puis à le réouvrir. Elle se révèle particulièrement utile pour la gestion d'erreurs.

Faire appel aux fonctions handle

Nous l'avons dit, le Pascal ne permet pas de récupérer les handles des fichiers gérés avec son aide. Il faut donc re-programmer toutes les opérations fichiers en Pascal si l'on veut pouvoir bénéficier des avantages procurés par les handles et nous verrons qu'ils sont nombreux.

Pour cela, il suffit d'écrire une unité prenant en charge toutes les opérations fichiers offertes par l'Int `21h` du DOS. Nous l'avons appelée `FHandle.Pas`. Plutôt que des références croisées, nous vous en présentons ici l'interface (le code se trouve en annexe) :

Listing 8.10
Interface de l'unité FHandle.Pas. ❶

```
UNIT FHandle;                          { FHandle.Pas }

INTERFACE

USES Dos;

CONST                        { Dispositifs standards d'E/S }
 InputDos  : Word = 0; { Con }
 OutputDos : Word = 1; { Con }
 ErrDos    : Word = 2; { Con }
 AuxDos    : Word = 3; { Aux  = Com1 }
 PrnDos    : Word = 4; { Prn  = Lpt1 }

TYPE Handle = Word;
     StrAsciiZ = ARRAY[0..255] OF Char;
     RecSearchFic = RECORD
                      Reserve      : ARRAY[0..$14] OF Byte;
                      Attr         : Byte;
                      Heure, Date  : Word;
                      Taille       : LongInt;
                      NomExt       : ARRAY[0..11] OF Char;
                    END;
 Tab = Pointer;

VAR Erreur, ErreurDos : Word;
    RecSearch         : RecSearchFic;

FUNCTION CreeFichier(Attr : Byte;
                     VAR Nom : StrAsciiZ) : Handle;

FUNCTION OuvreFichier(Mode : Byte;
                      VAR Nom : StrAsciiZ) : Handle;
```

Interface de l'unité FHandle.Pas (suite). ❷

```
PROCEDURE FermeFichier(VAR h : Handle);

FUNCTION LitFichierLogique(h : Handle; Nb : Word;
                           VAR Buf : Tab) : Word;

FUNCTION EcritFichierLogique(h : Handle; Nb : Word;
                             VAR Buf : Tab) : Word;

FUNCTION EffaceFic(VAR Nom : StrAsciiZ) : Word;

FUNCTION FSeek(Methode : Byte; h : Handle;
               Position : LongInt) : LongInt;

FUNCTION ChangeAttrFic(Attr : Byte;
                       VAR Nom : StrAsciiZ) : Word;

FUNCTION DonneAttrFic(VAR Nom : StrAsciiZ) : Word;

FUNCTION NewHandle(h : Handle) : Word;

FUNCTION RedirigeHandle(VAR hSrce, hDest : Handle) : Word;

PROCEDURE SetDTA(VAR Rec : RecSearchFic);

FUNCTION FindFirstFic(Attr : Byte;
                      VAR Nom : StrAsciiZ) : Word;

FUNCTION FindNextFic : Word;

FUNCTION RenFic(VAR NomActuel,
                NouveauNom : StrAsciiZ) :  Word;

FUNCTION ChangeDateHeure(h : Handle;
                         Date, Heure : Word) : Word;
FUNCTION DonneDateHeure(h : Handle;
                        VAR Date, Heure : Word) : Word;

FUNCTION CreeTemporaire(Attr : Byte;
                        VAR Nom : StrAsciiZ) : Word;

FUNCTION CreeNouveauFic(Attr : Byte;
                        VAR Nom : StrAsciiZ) : Word;

FUNCTION FixeMaxHandles(Nb : Word) : Word;

FUNCTION FlusheBufferFic(h : Handle) : Word;

IMPLEMENTATION
```

La quasi-totalité des opérations DOS a été implémentée sous forme de fonction. La principale raison tient à ce qu'une fonction est plus rapide qu'une procédure. Ceci est vrai dans tous les langages : il est plus facile d'assigner une valeur à un morceau de code (ce qui se fait par l'intermédiaire d'un pointeur) que de créer une variable en mémoire, y stocker le résultat et la renvoyer. La plupart de ces fonctions renvoient un handle. Mais ce n'est pas toujours le cas : méfiez-vous de celles qui retournent un nombre d'octets, ou une valeur 0 signifiant que tout va bien.

D'autre part, la variable `Erreur` contient toujours le résultat de la fonction : `0` si tout s'est bien passé, un code d'erreur sinon. La variable `ErreurDos` ne sert que s'il n'y a plus de fichiers correspondant aux paramètres passés à la fonction `TrouvePremierFic` dans le répertoire. Elle correspond à la variable `DosError` du Turbo Pascal.

Dans le même ordre d'idées, la variable `RecSearch` a exactement la même structure que l'enregistrement (`Record`) `SearchRec` de l'unité `DOS` du Turbo Pascal. C'est d'ailleurs normal puisque cette structure est imposée par le `DOS`. La seule différence tient aux chaînes de caractères représentant le nom du fichier, que nous avons laissé sous la forme ASCIIZ au lieu des `STRING` du Turbo Pascal.

Les types définis sont bien sûr `Handle`, un simple `Word`, `StrAsciiZ`, un tableau de 256 caractères commençant à l'index zéro, `RecSearchFic`, qui correspond à la variable `RecSearch` et `Tab`, un pointeur. Il nous a paru pratique de définir un pointeur pour les opérations de lecture et d'écriture sur fichier logique, plutôt que de passer à ces fonctions une variable sans type. Cela permet entre autres de charger un fichier d'un peu moins de 64 Ko (65 521 octets exactement) en une seule fois. Au delà, il faut bien entendu gérer les restes, mais c'est bien moins complexe que lorsque l'on charge un fichier secteur par secteur ou cluster par cluster, et cela n'influe pas sur le reste de la mémoire allouée aux autres variables. Bien entendu, cela suppose de respecter certaines règles, qui sont essentiellement des règles d'écriture et de ne pas diminuer excessivement la taille du tas. Mais ce sont des méthodes peu employées, et il est toujours possible de revenir aux bons vieux tableaux de 1 024 octets.

Enfin, on notera les constantes représentant les cinq handles prédéfinis. Elles sont particulièrement utiles lorsque l'on veut gérer la redirection ou créer un filtre.

Le petit programme d'exemple qui suit utilise l'unité `FHandle.Pas` pour lister un fichier texte sur le périphérique de votre choix : écran, imprimante, ou fichier. Le fichier source peut être le clavier : toutefois, le mode clavier restant inchangé, le programme s'arrêterait dans ce cas au premier retour chariot ou après le 1 024e caractère tapé.

Sans utiliser vraiment les possibilités de redirection du DOS, l'appel aux fonctions de lecture et d'écriture d'un fichier logique redirige éventuellement les entrées et les sorties. Mais l'intérêt principal du programme réside tout de même dans la

gestion des handles, qui n'est pas plus complexe que celle des fichiers «classiques» du Turbo Pascal.

Listing 8.11
Programme Lister.Pas. ❶

```
PROGRAM ListeFichierTexte;       { Lister.Pas }

USES FHandle;

VAR Chaine : String;
    Chemin : StrAsciiZ;
    Source, Dest : Handle;
    Taille, Bide : LongInt;

 PROCEDURE StrToAsciiZ(VAR S : String;
                       VAR Res : StrAsciiZ);
 BEGIN
   FillChar(Res[0], SizeOf(Res), 0);
   Move(S[1], Res[0], Length(S));
 END;

 PROCEDURE Liste(hSource, hDest : Handle;
                 Taille : LongInt);

 CONST Erreur1 : String[80] =
 #13 + #10 + ' E R R E U R  en  lecture          '+
       '                                        ';
       Erreur2 : String[80] =
 #13 + #10 + ' E R R E U R  en  écriture         '+
       '                                        ';
       Erreur3 : String[80] =
 #13 + #10 + ' E R R E U R  :  fichier source trop grand '+
       '                                ';


 VAR Buf, Ch : Tab;
     Nb : Word;
     Err : Byte;
 BEGIN
   Err := 0;
   IF (Taille <= 65521) THEN
   BEGIN
     GetMem(Buf, Taille);
     Nb := LitFichierLogique(hSource, Taille, Buf);
     IF (Nb = Taille) THEN
```

➥

Programme Lister.Pas (suite). ❷

```
      BEGIN
        Nb := EcritFichierLogique(hDest, Taille, Buf);
        IF (Nb <> Taille) THEN
        BEGIN
          Err := 2;
          Ch := Addr(Erreur2);
          Nb := EcritFichierLogique(OutputDos, 80, Ch);
          IF (hDest > 4) THEN
            FermeFichier(hDest);
        END;
      END
      ELSE
      BEGIN
        Err := 1;
        Ch := Addr(Erreur1);
        Nb := EcritFichierLogique(OutputDos, 80, Ch);
        IF (hSource > 4) THEN
          FermeFichier(hSource);
      END;
      FreeMem(Buf, Taille);
      CASE Err OF
        1 : IF (hDest > 4) THEN
              FermeFichier(hDest);
        2 : IF (hSource > 4) THEN
              FermeFichier(hSource);
      ELSE
        BEGIN
          IF (hDest > 4) THEN
            FermeFichier(hDest);
          IF (hSource > 4) THEN
            FermeFichier(hSource);
        END;                          { Else Case }
      END;                            { Case }
    END
    ELSE
    BEGIN
      Ch := Addr(Erreur3);
      Nb := EcritFichierLogique(OutputDos, 80, Ch);
    END;
  END;


BEGIN
  IF (ParamCount < 1) THEN
```

➥

Programme Lister.Pas (suite). ❸

```
  BEGIN
    Source := InputDos; Dest := OutputDos;
  END
  ELSE
  IF (ParamCount = 1) THEN
  BEGIN
    Chaine := ParamStr(1);
    StrToAsciiZ(Chaine, Chemin);
    Source := OuvreFichier(0, Chemin); Dest := OutputDos;
  END
  ELSE
  IF (ParamCount = 2) THEN
  BEGIN
    Chaine := ParamStr(1); StrToAsciiZ(Chaine, Chemin);
    Source := OuvreFichier(0, Chemin);
    Chaine := ParamStr(2); StrToAsciiZ(Chaine, Chemin);
    Dest := OuvreFichier(1, Chemin);
  END;
  Taille := FSeek(2, Source, 0);
  Bide   := FSeek(0, Source, 0)
  IF (Taille = 0) THEN
    Taille := 1024;
  Liste(Source, Dest, Taille);
END.
```

Si ce programme est loin d'être exemplaire par sa gestion des erreurs et ses tests souvent répétitifs, il donne cependant une bonne idée de la programmation à base de handles.

Gestion interne des handles par MS-DOS

Après avoir entrevu les grands principes qui président à l'utilisation des handles et donné un exemple simple, nous nous intéressons maintenant à la façon dont le DOS gère les handles. Nous voilà donc revenus à la programmation système.

Jusqu'ici, nous ne savons que deux choses sur les handles et le DOS : il y a vingt handles disponibles par programme (dont les cinq standard), et un maximum de 255 en tout pour le système. En outre, si l'utilisateur n'a pas besoin d'en savoir plus sur un fichier que son numéro d'identification, le DOS, lui, doit pouvoir retrouver

le fichier pour le traiter. Cela suppose donc qu'il conserve les renseignements concernant chaque fichier quelque part dans ses structures de données.

La File Handle Table : table des handles

On l'aura compris, un handle n'est jamais qu'un pointeur sur les renseignements conservés en mémoire par le DOS sur chaque fichier. C'est d'ailleurs ce qui fait que l'on peut si aisément remplacer l'écran par l'imprimante ou par un fichier : il suffit d'échanger deux pointeurs en mémoire. Mais où les handles propres à un programme sont-ils conservés ? L'emplacement le plus logique serait le PSP : il y a un PSP par programme. Chacun peut donc conserver une table des handles utilisés. C'est exactement ce qui passe. Nous avions vu au chapitre 3 (*RAM gérée par le DOS*) qu'un PSP contenait plusieurs champs réservés. Parmi ceux-ci, trois sont dédiés à la gestion des handles du programme. Ce sont les champs d'offset `18h`, `32h` et `34h`. La table des handles de fichiers fait vingt octets de long et se trouve à l'offset `18h` dans le PSP. Le mot (`Word`) à l'adresse `32h` contient le nombre de handles utilisables par le PSP. Le double mot (`DWord`, `LongInt` en Pascal) de l'adresse `34h` donne l'adresse de la table des handles de fichiers, qui est normalement d'offset `18h`.

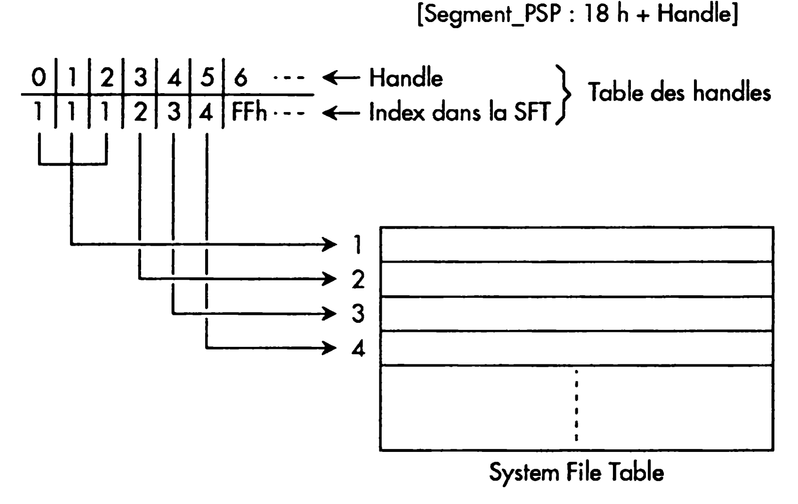


Figure 8.12
Handles, table des handles et table des fichiers du système.

Un handle n'est autre que le numéro de l'index de la table des handles. L'entrée correspondant à cet index dans la table des handles contient un octet qui désigne l'index auquel trouver le descriptif du fichier dans la table des fichiers du système. La table des fichiers du système est organisée en liste de plusieurs tableaux d'enregistrements qui pointent les uns sur les autres. On a donc affaire à un

ensemble de pointeurs : le handle pointe sur l'entrée de la table des handles, dont le contenu pointe sur un pointeur qui pointe la première entrée de la table des fichiers du système (voir *figure 8.12 ci-dessus*).

Chaque entrée de la table des handles contient l'index auquel se trouve le descriptif du fichier ou du périphérique correspondant au handle. Plusieurs handles peuvent désigner le même descriptif. Une entrée inutilisée est signalée par la valeur `FFh`.

N'importe quel programme peut donc accéder à sa table des handles : il lui suffit de lire les valeurs qui se trouvent en `Seg(PSP) :34h` et en `Seg(PSP) :32h`. Il lit ensuite autant d'octets qu'il y a de handles dans la table et peut déterminer si la table est remplie ou non ou s'il faut la remplacer par une autre. Il peut même, en accédant ensuite à la table des fichiers du système, connaître les caractéristiques de chacun des fichiers pointés et déceler – ou provoquer – d'éventuelles redirections.

```
PROGRAM LitFileHandleTable;        {LitFHT.pas}



PROCEDURE LitFHT;
VAR AdrPSP, AdrOfsFHT, AdrSegFHT, NbHdl : Word;
    i                                   : Byte;

BEGIN
  AdrPSP := PrefixSeg;
  AdrOfsFHT := MemW[AdrPSP:$34];
  AdrSegFHT := MemW[AdrPSP:$36];
  NbHdl := MemW[AdrPSP:$32];
  FOR i := 0 TO NbHdl-1 DO
  BEGIN
    Write(Mem[AdrSegFHT:AdrOfsFHT+i], ' ');
  END;
END;

BEGIN
  LitFHT;
END.
```

Encadré 8.13
Accéder à la table des handles du PSP courant.

La System File Table : table des fichiers

La taille de la table des fichiers du système est déterminée au lancement du micro par la lecture du fichier `CONFIG.SYS`. Cette table contient autant d'enregistrements qu'il y a de fichiers déclarés dans la directive «`FILES = nnn`». La table des fichiers renseigne le DOS sur l'état actuel des fichiers aussi bien que sur celui des pilotes de

périphériques. Un même enregistrement peut se trouver pointé par plusieurs entrées de la table des handles d'un PSP. Le format d'une entrée de la table des fichiers du système diffère selon que l'entrée concerne un fichier ou un pilote de périphérique. Il y a une table des fichiers système pour les FCB et une autre pour les handles. Chacune a le même format qui est examiné au *tableau 8.14*.

Format de l'en-tête d'un élément de la SFT

Adresse	*Signification*
00h	Prochain élément
04h	Nombre d'entrées dans l'élément

Format d'une entrée d'un élément de la SFT

Adresse	*Signification*
00h	Nombre d'ouvertures précédentes
02h	Mode d'ouverture :

Valeurs	*Signification*
8000h	FCB
0040h	Bits de partage : accepter tous les accès
0030h	Accepter la lecture
0020h	Accepter l'écriture
0010h	Tout refuser
0070h	FCB en réseau
0001h	Bits d'accès : fichier en écriture
0000h	Fichier en lecture

Adresse	*Signification*
04h	Attribut
05h	Drapeaux :

Valeurs	*Signification*
8000h	Accès en réseau
4000h	Inscrire la date (fichiers)
4000h	Supporte IOCTL (périphériques)

(suite du tableau)

Adresse	Valeurs	Signification
	0080h	Entrée pour périphérique
	0040h	Fin de fichier en entrée (périphériques)
	0020h	Mode d'écriture binaire (périphériques)
	0010h	Spécial : le périphérique supporte les sorties de l'Int 29h
	0008h	Périphérique horloge courant
	0004h	Périphérique NUL
	0002h	Périphérique de sortie courant
	0001h	Périphérique d'entrée courant
	0040h	Fichier écrit
	003Fh	Masque pour les bits de lecteur
07h	Adresse du premier DBP (Disk Block Parameters)	
0Bh	Premier cluster	
0Dh	Heure	
0Fh	Date	
11h	Taille du fichier	
15h	Position courante dans le fichier	
19h	Cluster relatif au début du fichier	
1Bh	Cluster courant	
1Dh	Numéro de bloc	
1Fh	Index dans le répertoire	
20h	Nom et extension du fichier	
2Bh	Signification inconnue	
2Fh	Numéro d'identification du PC du propriétaire du fichier	
31h	Propriétaire du fichier (PSP ou premier programme à y avoir accédé)	
33h	Etat du fichier	

Tableau 8.14
Format de la table des fichiers du système.

Les deux tableaux de la *figure 8.14* montrent le format de l'en-tête d'un élément de la SFT et celui d'une entrée dans un élément. La signification des quatre octets qui se trouvent à l'offset 2Bh est restée inconnue : la table des fichiers du système n'est en effet pas documentée par Microsoft.

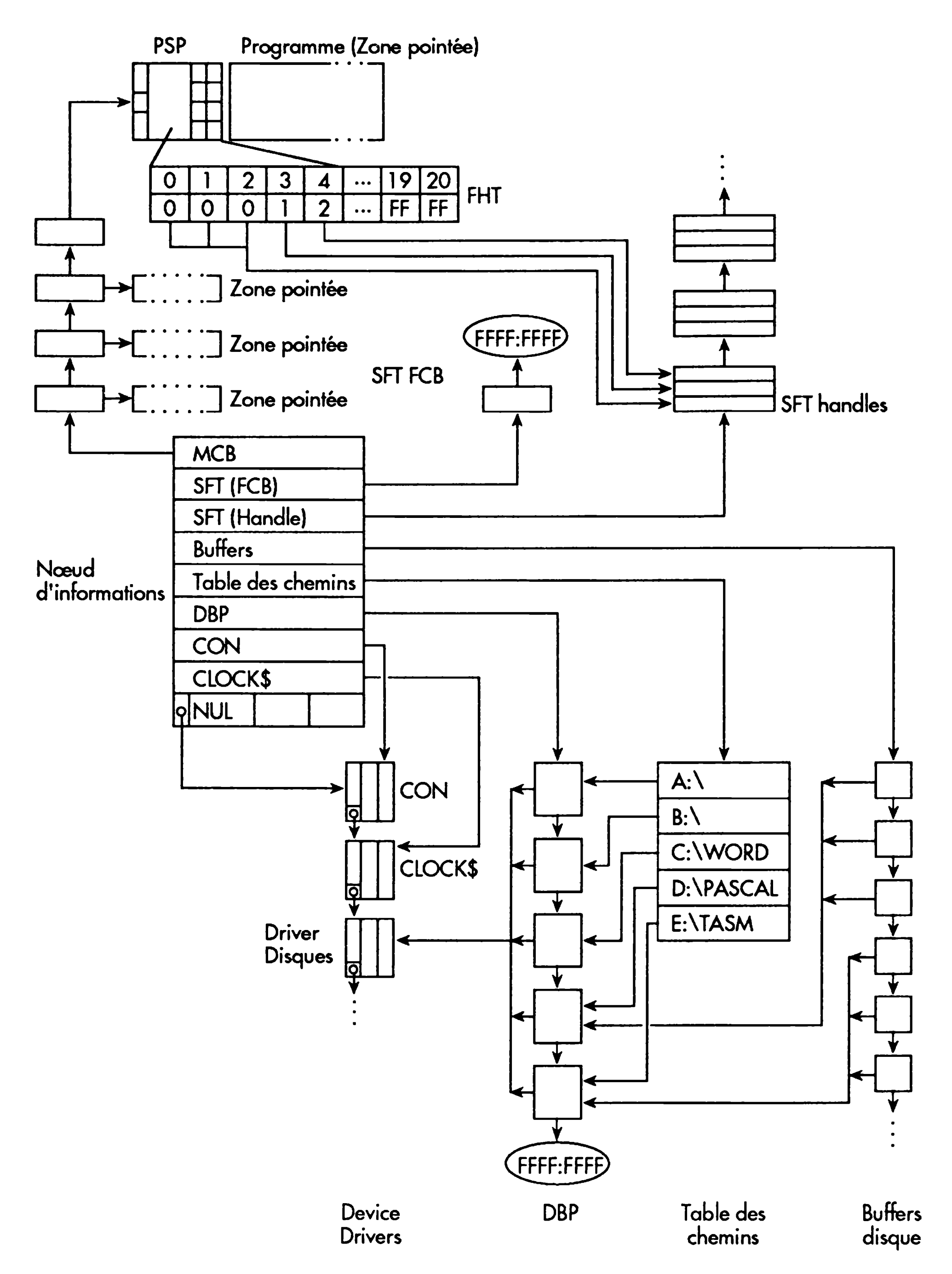


Figure 8.15
Le nœud d'informations du DOS au cœur du système.

Le premier champ de l'en-tête est un pointeur sur le prochain en-tête. S'il contient la valeur FFFFh:FFFFh, il s'agit du dernier en-tête. Le mot d'offset 04h dans l'en-tête de bloc contient le nombre d'entrées du bloc pointé. Le mot d'offset 00h dans l'entrée de la table des fichiers se rapporte au nombre de fois que le fichier pointé a été ouvert : un même fichier pouvant être ouvert par plusieurs programmes, le DOS n'alloue pas à chaque fois une nouvelle entrée, mais incrémente ce compteur. Lorsqu'il ferme le fichier, il le décrémente, mais ne libère l'entrée que lorsque le compteur est mis à zéro. Les différentes valeurs du mode d'ouverture du fichier, ainsi que celles des drapeaux, peuvent se combiner –ainsi les bits de partage et les bits d'accès au fichier. Les blocs de contrôle du disque (DBP, *Disk block parameters*) ont été vus au chapitre 3 (*RAM gérée par le DOS*).

On accède aux deux tables des fichiers du système par l'intermédiaire de la fonction 52h de l'Int 21h. Celle-ci renvoie en ES:BX+4 un pointeur sur la table handle des fichiers du système et en ES:BX+20h un pointeur sur la table FCB. Si le fichier CONFIG.SYS ne contient pas de directive «FCB=nnn», l'adresse renvoyée est fausse et le système ne contient pas de table FCB. Nous sommes maintenant à même de comprendre comment fonctionne exactement le nœud d'informations du DOS, dont nous avions déjà parlé au chapitre 3 (*RAM gérée par le DOS*).

Accéder à la System File Table

Le programme SFT.PAS permet de lire la table des handles d'un PSP et d'accéder ensuite à l'enregistrement de la System File Table correspondant au handle que l'on a sélectionné. On peut examiner soit le PSP du programme, soit n'importe lequel dont on aura donné l'adresse de segment en paramètre. Aucune vérification n'étant effectuée sur la validité de l'adresse du PSP, on aura tout intérêt à la vérifier auparavant en lançant le programme MapPSP.Pas (voir chapitre 3, *RAM gérée par le DOS*). Les vingt premiers enregistrements de la FHT apparaissent ensuite à l'écran : il suffit d'en choisir un en déplaçant le curseur pour obtenir l'interprétation de l'enregistrement de la SFT qu'il pointe.

Le défaut principal de ce programme, auquel il est facile de remédier, tient à ce qu'il visualise uniquement les handles par défaut de la FHT. En effet, les fichiers ouverts par un programme sont automatiquement refermés dès que le processus a pris fin. Dès lors, les handles sont libérés et les entrées de la FHT contiennent la valeur FFh. Si l'on souhaite se libérer de cette limitation, il suffit de programmer quelques lignes demandant à l'utilisateur de préciser le nom des fichiers à ouvrir. Le programme ouvrirait ensuite ces fichiers avec les procédures de l'unité FHandle.Pas et continuerait de se dérouler normalement. Dans un tel cas de figure, il faudrait que le PSP choisi soit celui de SFT.Pas et que les fichiers soient refermés à la fin du programme. La taille déjà importante du programme (trop grande pour le livre) nous a dissuadé d'ajouter nous-mêmes ces quelques lignes.

Nom	*Type*	*Description*	*Ligne*
`OfsFHT`	F	Fournit l'offset de la File Handle Table du PSP passé en paramètre	*40*
`SegFHT`	F	Fournit le segment de la FHT du PSP passé en paramètre	*45*
`NbEntreesFHT`	F	Renvoie le nombre d'entrées que contient la FHT	*50*
`CopieFHT`	P	Copie la FHT dans un tableau de 20 octets	*55*
`NbNosFHTDispo`	F	Donne le nombre de handles utilisés dans la File Handle Table	*61*
`SegSFT`	F	Fournit le segment de la System File Table	*73*
`OfsSFT`	F	Fournit l'offset de la System File Table	*91*
`ChercheEntree`	F	Met à jour le record contenant les champs de l'entrée de la FHT pointée par le numéro de la FHT	*109*
`\Groupe`	F	Renvoie le groupe de la SFT auquel appartient l'entrée de la SFT	*120*
`EcranSFT`	P	Affiche les champs du Record contenant l'entrée de la SFT	*198*
`\Zero`	F	Renvoie une chaîne formatée avec un zéro (date ou heure)	*204*
`Ecran`	P	Affiche l'écran	*280*
`Deplace`	P	Déplacement dans la FHT	*300*
`\Curseur`	P	Affiche le curseur en surimpression ou l'efface	*304*

Dépendances :

Procédure	*Procédure appelée*
`Programme`	`OfsFHT`
	`SegFHT`
	`NbEntreesFHT`
	`CopieFHT`
	`Ecran`
	`Deplace`
`Ecran`	`OfsFHT`
	`SegFHT`
`ChercheEntree`	`ChercheEntree\Groupe`
`\Groupe`	`SegSFT`
	`OfsSFT`

➥

(suite du tableau)

Procédure	**Procédure appelée**
EcranSFT	ChercheEntree
	EcranSFT\Zero
Deplace	EcranSFT
	Ecran
	Deplace\Curseur

Tableau 8.16
Références croisées de SFT.Pas.

Listing 8.17
Programme SFT.Pas. ❶

```
PROGRAM ExamenDeLaSystemFileTable; { SFT.Pas }

USES  Dos, Crt, Sys;

TYPE  FileHandleTable = ARRAY[1..20] OF Byte;
      StrAsciiZ       = ARRAY[0..255] OF Char;
      EnTeteSFT       = RECORD
                          Suivant   : Pointer;
                          NbEntrees : Word;
                        END;        { Record }
      EntreeSFT       = RECORD
                          NbOpen,
                          OpenMode : Word;
                          Attribut : Byte;
                          Flags    : Word;
                          DPB1     : Pointer;
                          Clust1,
                          Heure,
                          Date     : Word;
                          Taille,
                          Position : LongInt;
                          ClRelatif,
                          ClCourant,
                          NoBloc   : Word;
                          IdxRep   : Byte;
                          Nom      : StrAsciiZ;
                          Inconnu  : LongInt;
                          NoPC,
                          Pere,
                          EtatFic  : Word;
                        END;       { Record }
```

➥

Programme SFT.Pas (suite). ❷

```
VAR  i, ErreurDos,
     Grpe, PSPSeg : Word;
     Tab          : FileHandleTable;
     Entree       : EntreeSFT;
     Sauve        : Byte;
     S            : String;

 FUNCTION OfsFHT(SegPSP : Word) : Word;
 BEGIN
   OfsFHT := MemW[SegPSP:$34];
 END;

 FUNCTION SegFHT(SegPSP : Word) : Word;
 BEGIN
   SegFHT := MemW[SegPSP:$36];
 END;

 FUNCTION NbEntreesFHT(FHTSeg : Word) : Word;
 BEGIN
   NbEntreesFHT := MemW[FHTSeg:$32];
 END;

 PROCEDURE CopieFHT(FHTSeg, FHTOfs, Nb : Word;
                    VAR Tab : FileHandleTable);
 BEGIN
   FillChar(Tab, SizeOf(Tab), 0);
   Move(Mem[FHTSeg:FHTOfs], Mem[Seg(Tab):Ofs(Tab)], Nb);
 END;

 FUNCTION NbNosFHTDispo(FHTSeg, FHTOfs : Word) : Word;
 VAR i : Word;
 BEGIN
   i := 0;
   WHILE (Mem[FHTSeg:FHTOfs + i] < $FF) DO
     Inc(i);
   IF ((Mem[FHTSeg:FHTOfs + (i - 1)] <> Tab[i]) OR
      (Mem[FHTSeg:FHTOfs + i] <> Tab[i + 1])) THEN
     CopieFHT(FHTSeg, FHTOfs, NbEntreesFHT(FHTSeg), Tab);
   NbNosFHTDispo := i;                 { à partir de 1 }
 END;

 FUNCTION SegSFT : Word;
 VAR Regs : Registers;
```

➥

Programme SFT.Pas (suite). ❸

```
  BEGIN
    WITH Regs DO
    BEGIN
      Ah := $52;
      MsDos(Regs);
      IF (Flags AND 1 = 1) THEN
      BEGIN
        ErreurDos := Ax; SegSFT := $FFFF;
      END
      ELSE
      BEGIN
        ErreurDos := 0; SegSFT := MemW[Es:(Bx + 6)];
      END;
    END;
  END;

  FUNCTION OfsSFT : Word;
  VAR Regs : Registers;
  BEGIN
    WITH Regs DO
    BEGIN
      Ah := $52;
      MsDos(Regs);
      IF (Flags AND 1 = 1) THEN
      BEGIN
        ErreurDos := Ax; OfsSFT := $FFFF;
      END
      ELSE
      BEGIN
        ErreurDos := 0; OfsSFT := MemW[Es:(Bx + 4)];
      END;
    END;
  END;

 FUNCTION ChercheEntree(NoFHT : Word) : Word;
  CONST  TailleEnTete : Byte = 6;
         TailleEntree : Byte = $35;

  VAR SFTSeg, SFTOfs, No,
      OfsEntree            : Word;
      EnTete               : EnTeteSFT;
      Tempo, Tempo2        : LongInt;
      Premier              : Pointer;
```

➥

Programme SFT.Pas (suite). ❹

```
  FUNCTION Groupe(VAR NoFHT, Grpe : Word) : Pointer;
  VAR SFTSeg, SFTOfs,
      SegSvt, OfsSvt, NbE : Word;
      Premier, Prochain   : Pointer;
  BEGIN
    SFTSeg := SegSFT; SFTOfs := OfsSFT;
    Premier := Ptr(SFTSeg, SFTOfs);
    NbE := MemW[SFTSeg:SFTOfs + 4];
    SegSvt := MemW[Seg(Premier^):Ofs(Premier^) + 2];
    OfsSvt := MemW[Seg(Premier^):Ofs(Premier^)];
    Prochain := Ptr(SegSvt, OfsSvt); Grpe := 0;
    WHILE ((NoFHT > NbE) AND ((Seg(Premier^) <> $FFFF) OR
                          (Ofs(Premier^) <> $FFFF))) DO
    BEGIN
      Premier := Prochain;
      SegSvt := MemW[Seg(Premier^):Ofs(Premier^) + 2];
      OfsSvt := MemW[Seg(Premier^):Ofs(Premier^)];
      Prochain := Ptr(SegSvt, OfsSvt); Dec(NoFHT, NbE);
      NbE := MemW[Seg(Premier^):Ofs(Premier^) + 4];
      Inc(Grpe);
    END;
    WITH EnTete DO
    BEGIN
      Suivant := Prochain;
      NbEntrees := NbE;
    END;
    Groupe := Premier;
  END;

BEGIN          {CherceEntrées}
  No := NoFHT;
  Premier := Groupe(No, Grpe);
  SFTSeg := Seg(Premier^); SFTOfs := Ofs(Premier^);
  OfsEntree := SFTOfs + TailleEnTete +
               (TailleEntree * No);
  IF (EnTete.NbEntrees > NoFHT) THEN
  BEGIN
    WITH Entree DO
    BEGIN
      NbOpen := MemW[SFTSeg:OfsEntree];
      OpenMode := MemW[SFTSeg:OfsEntree + 2];
      Attribut := Mem[SFTSeg:OfsEntree + 4];
      Flags := MemW[SFTSeg:OfsEntree + 5];
               MemW[SFTSeg:OfsEntree + 7]);
```

➥

Programme SFT.Pas (suite). ❺

```
DPB1 := Ptr(MemW[SFTSeg:OfsEntree + 9],
            MemW[SFTSeg:OfsEntree + 7]);
Clust1 := MemW[SFTSeg:OfsEntree + $B];
Heure := MemW[SFTSeg:OfsEntree + $D];
Date := MemW[SFTSeg:OfsEntree + $F];
Tempo := Mem[SFTSeg:OfsEntree + $14];
Tempo := Tempo SHL 24;
Tempo2 := Mem[SFTSeg:OfsEntree + $13];
Tempo2 := Tempo2 SHL 16;
Inc(Tempo, Tempo2);
Tempo2 := Mem[SFTSeg:OfsEntree + $12];
Tempo2 := Tempo2 SHL 8;
Inc(Tempo, Tempo2);
Inc(Tempo, Mem[SFTSeg:OfsEntree + $11]);
Taille := Tempo;
Tempo := Mem[SFTSeg:OfsEntree + $18];
Tempo := Tempo SHL 24;
Tempo2 := Mem[SFTSeg:OfsEntree + $17];
Tempo2 := Tempo2 SHL 16;
Inc(Tempo, Tempo2);
Tempo2 := Mem[SFTSeg:OfsEntree + $16];
Tempo2 := Tempo2 SHL 8;
Inc(Tempo, Tempo2);
Inc(Tempo, Mem[SFTSeg:OfsEntree + $15]);
Position := Tempo;
ClRelatif := MemW[SFTSeg:OfsEntree + $19];
ClCourant := MemW[SFTSeg:OfsEntree + $1B];
NoBloc := MemW[SFTSeg:OfsEntree + $1D];
IdxRep := Mem[SFTSeg:OfsEntree + $1F];
Move(Mem[SFTSeg:OfsEntree + $20], Nom[0], $B);
Tempo := Mem[SFTSeg:OfsEntree + $2E];
Tempo := Tempo SHL 24;
Tempo2 := Mem[SFTSeg:OfsEntree + $2D];
Tempo2 := Tempo2 SHL 16;
Inc(Tempo, Tempo2);
Tempo2 := Mem[SFTSeg:OfsEntree + $2C];
Tempo2 := Tempo2 SHL 8;
Inc(Tempo, Tempo2);
Inc(Tempo, Mem[SFTSeg:OfsEntree + $2B]);
Inconnu := Tempo;
NoPC := MemW[SFTSeg:OfsEntree + $2F];
Pere := MemW[SFTSeg:OfsEntree + $31];
```

➥

Programme SFT.Pas (suite). ❻

```
      EtatFic := MemW[SFTSeg:OfsEntree + $33];
    END;
  END;
  ChercheEntree := No;
END;

PROCEDURE EcranSFT(NoFHT : Word);
VAR No   : Word;
    HetD : LongInt;
    R    : DateTime;
    S    : String;

  FUNCTION Zero(No : Word) : String;
  VAR S : String;
  BEGIN
    Str(No, S);
    IF No < 10 THEN
      S := '0' + S;
    Zero := S;
  END;

BEGIN           {Ecran SFT}
  No := ChercheEntree(NoFHT);
  TextAttr := 15+4*16; GotoXy(37, 1); Write(' S F T ');
  GotoXy(27, 3); Write(' Entrée n° ', No:2);
  Write(' Groupe n° ', Grpe, ' ');
  GotoXy(4, 5); Write(' Nbre d''ouvertures     : ');
  GotoXy(4, 6); Write(' Mode d''ouverture      : ');
  GotoXy(4, 7); Write(' Attribut              : ');
  GotoXy(4, 8); Write(' Drapeaux              : ');
  GotoXy(4, 9); Write(' Bloc de param. disque : ');
  GotoXy(4, 10); Write(' Premier cluster      : ');
  GotoXy(4, 11); Write(' Heure                : ');
  GotoXy(4, 12); Write(' Date                 : ');
  GotoXy(4, 13); Write(' Taille               : ');
  GotoXy(4, 14); Write(' Position             : ');
  GotoXy(4, 15); Write(' Cluster relatif      : ');
  GotoXy(4, 16); Write(' Cluster courant      : ');
  GotoXy(4, 17); Write(' Numéro de bloc       : ');
  GotoXy(4, 18); Write(' Index dans répertoire : ');
  GotoXy(4, 19); Write(' Nom                  : ');
  GotoXy(4, 20); Write(' Inconnu (LongInt)    : ');
  GotoXy(4, 21); Write(' Numéro du PC         : ');
  GotoXy(4, 22); Write(' Propriétaire         : ');
  GotoXy(4, 23); Write(' Etat du fichier      : ');
  TextAttr := 15 + 3 * 16;
```

➥

Programme SFT.Pas (suite). ❼

```
    WITH Entree DO
    BEGIN
      GotoXy(29, 5); Write('':15, NbOpen:2, ' ');
      GotoXy(29, 6); Write(MotDecVersBin(OpenMode), ' ');
      GotoXy(29, 7); Write(MotDecVersBin(Attribut), ' ');
      GotoXy(29, 8); Write('':13,MotDecVersHex(Flags),' ');
      GotoXy(29, 9); Write('':8, MotDecVersHex(Seg(DPB1^)),
                     ':', MotDecVersHex(Ofs(DPB1^)), ' ');
      GotoXy(29, 10);
      Write('':13, MotDecVersHex(Clust1),' ');
      HetD := (Date SHL 16) + Heure;
      UnpackTime(HetD, R);
      GotoXy(29, 11); Write('':9, Zero(R.Hour), ':',
                       Zero(R.Min), ':', Zero(R.Sec),' ');
      GotoXy(29, 12); Write('':7, Zero(R.Day), '/',
                   Zero(R.Month), '/', Zero(R.Year), ' ');
      GotoXy(29, 13); Write('':7, Taille:10, ' ');
      GotoXy(29, 14); Write('':7, Position:10, ' ');
      GotoXy(29, 15);
      Write('':13, MotDecVersHex(ClRelatif),' ');
      GotoXy(29, 16);
      Write('':13, MotDecVersHex(ClCourant), ' ');
      GotoXy(29, 17); Write('':10, NoBloc:7, ' ');
      GotoXy(29, 18); Write('':14, IdxRep:3, ' ');
      GotoXy(29, 19); i := 0; S := '';
      WHILE (i <= 255) DO
      BEGIN
        IF (Nom[i] > #0) THEN
          S := S + Nom[i]
        ELSE
          i := 255;
        Inc(i);
      END;
      IF (S[9] <> ' ') THEN
        S := Copy(S, 1, 8) + '.' + Copy(S, 9, 3)
      ELSE
        S := Copy(S, 1, Pos(' ', S) - 1);
      Write('':(18 - (Length(S) + 1)), S, ' ');
      GotoXy(29, 20); Write('':7, Inconnu:10, ' ');
      GotoXy(29, 21); Write('':10, NoPC:7, ' ');
      GotoXy(29, 22); Write('':13,MotDecVersHex(Pere),' ');
      GotoXy(29, 23); Write('':10, EtatFic:7, ' ');
    END;
    ReadLn;
  END;
```

➥

Programme SFT.Pas (suite). ❽

```
PROCEDURE Ecran(PSPSeg : Word);
VAR i                : Byte;
    FHTSeg, FHTOfs : Word;
BEGIN
  FHTSeg := SegFHT(PSPSeg); FHTOfs := OfsFHT(PSPSeg);
  TextAttr := 15 + 1 * 16; ClrScr;
  TextAttr := 15 + 4 * 16; GotoXy(37, 1);
  Write(' F H T '); GotoXy(5, 3); Write('PSP en : ');
  GotoXy(60, 3); Write('FHT en : '); GotoXy(10, 7);
  FOR i := 0 TO 19 DO
    Write(i:2, ' ');
  TextAttr := 14 + 1 * 16; GotoXy(15, 3);
  Write(MotDecVersHex(PSPSeg), ':0000');
  GotoXy(70, 3); Write(MotDecVersHex(FHTSeg), ':',
                       MotDecVersHex(FHTOfs));
  GotoXy(11, 8);
  FOR i := 1 TO 20 DO
    Write(OctetDecVersHex(Tab[i]), ' ');
END;

PROCEDURE Deplace(PSPSeg : Word);
VAR Col, No : Byte;
    Car     : Char;

  PROCEDURE Curseur(Col, No, Attr : Byte);
  BEGIN
    GotoXy(Col, 7); TextAttr := Attr; Write(No:2, ' ');
  END;

BEGIN    {Déplace}
  Col := 10; Car := #215; No := 0;
  Curseur(Col, No, 15 + 3 * 16);
  WHILE (Car <> #27) DO
  BEGIN
    Car := ReadKey;
    CASE Car OF
      #0 : BEGIN
             Car := ReadKey;
             Curseur(Col, No, 15 + 4 * 16);
             CASE Car OF
               #77 : BEGIN
                       Inc(Col, 3); Inc(No);
                     END;
```

➥

Programme SFT.Pas (suite). ❾

```
                #75 : BEGIN
                        Dec(Col, 3); Dec(No);
                      END;
                #71 : BEGIN
                        No := 0; Col := 10;
                      END;
                #79 : BEGIN
                        No := 19; Col := 67;
                      END;
              END;                    { Case }
              IF (Col > 67) THEN
              BEGIN
                No := 0; Col := 10;
              END
              ELSE
              IF (Col < 10) THEN
              BEGIN
                No := 19; Col := 67;
              END;
              Curseur(Col, No, 15 + 3 * 16);
            END;
      #13 : BEGIN
               IF (Tab[No + 1] <> $FF) THEN
               BEGIN
                 TextAttr := 15 + 1 * 16; ClrScr;
                 EcranSFT(Tab[No + 1]);
                 Ecran(PSPSeg);
                 Curseur(Col, No, 15 + 3 * 16);
               END
               ELSE
               BEGIN
                 GotoXy(12, 15); TextAttr := 4 + 7 * 16;
                 Write(' E R R E U R : Impossible de ' +
                'voir un handle non affecté ');
                 ReadLn;
                 GotoXy(1, 15); TextAttr := 15 + 1 * 16;
                 Write('':79);
               END;
             END;
    END;                              { Case }
  END;
END;
```

Programme SFT.Pas (suite). 10

```
BEGIN
 Sauve := TextAttr;
 IF (ParamCount < 1) THEN
   PSPSeg := PrefixSeg
 ELSE
 BEGIN
   S := ParamStr(1);
   FOR i := 1 TO Length(S) DO
     S[i] := Upcase(S[i]);
   PSPSeg := HexaVersDecimal(S);
 END;
 CopieFHT(SegFHT(PSPSeg), OfsFHT(PSPSeg),
          NbEntreesFHT(SegFHT(PSPSeg)), Tab);
 Ecran(PSPSeg);
 Deplace(PSPSeg);
 TextAttr := Sauve; ClrScr;
END.
```

Filtres et redirection

La *redirection* est une des spécificités que le DOS a empruntée à UNIX. On appelle redirection la possibilité de détourner soit les sorties soit les entrées, ou encore les sorties et les entrées, d'un programme. Lorsque l'utilisateur tape la commande :

«`TYPE Fichier.Txt > Prn`»,

il redirige la sortie de la commande `TYPE` sur l'imprimante. Le résultat est que le fichier s'imprimera sur papier au lieu de s'afficher à l'écran. Deux fonctions de l'Int `21h` (`45h` et `46h`) permettent de gérer la redirection «à la main». Mais le DOS s'en occupe très bien lui-même.

Les *filtres* sont des programmes qui utilisent intensivement les possibilités de redirection mises à leur disposition par le DOS. L'utilitaire `MORE` du DOS est un filtre : si on l'appelle sans lui passer d'argument, il affichera page par page les données tapées au clavier (entrée standard) sur l'écran (sortie standard). Mais si l'on tape :

«`MORE < Fichier.Txt`»,

`Fichier.Txt` deviendra l'entrée standard, qu'il affichera à l'écran page par page. On peut aussi taper la commande :

«`MORE < Fichier.Txt > Prn`»,

et l'imprimante deviendra la sortie standard. Enfin, et c'est le plus intéressant, on peut également donner au programme MORE les sorties du programme TYPE (par exemple), pour qu'il en fasse ses propres entrées : cet enchaînement est appelé *piping*. C'est ce que fait la commande :

«TYPE Fichier.Txt | MORE».

Il va de soi que l'on peut compliquer encore un peu l'affaire et taper :

«DIR *.* /w | SORT /+9 | MORE > Fichier.Txt»,

ce qui trie les fichiers du répertoire courant par extension et les écrit sur 5 colonnes de 25 lignes dans un fichier (si vous essayez de passer cette commande, l'ordre dans lequel apparaîtront les noms de fichiers à l'écran est quelque peu perturbé par la commande de tri).

Les filtres peuvent se révéler très utiles : les utilitaires FIND et FC, fournis avec le DOS, rendent bien plus de services si leur sortie est redirigée sur un fichier ou l'imprimante, que si les différences trouvées sont affichées à l'écran. Mais leur principal intérêt est sans doute qu'ils sont faciles à programmer.

Un filtre n'a en effet d'autre rôle que de recopier ce qu'il lit sur l'entrée standard vers la sortie standard : c'est le DOS qui s'occupe de gérer les redirections. Le filtre agit donc en deux temps :

1. il lit une certaine quantité de données sur l'entrée standard à l'aide de la fonction 3Fh de l'Int 21h ;
2. il écrit sur la sortie standard les données qu'il a lues à l'aide de la fonction 40h de l'Int 21h.

Si la ligne de commande du filtre contient des opérateurs de redirection, le DOS substitue aux handles d'entrée et de sortie standard ceux des fichiers ou des périphériques les remplaçant. Le filtre lui-même lit donc bien l'enregistrement de la SFT correspondant aux handles 0 (clavier) et 1 (écran). Mais comme le numéro d'enregistrement de ces handles a changé, les enregistrements eux-mêmes ne sont pas ceux du clavier et de l'écran mais bien ceux de l'entrée et de la sortie qui leur ont été substitués. Autrement dit, pour écrire un filtre, il suffit de laisser le DOS faire son propre travail tout seul.

Le filtre que nous vous proposons (Filtre.Pas) cherche une chaîne de caractères passée en paramètre dans le fichier d'entrée et inscrit sur le fichier de sortie les adresses d'offset où elle a été trouvée.

Listing 8.18
Programme Filtre.Pas.

❶

```
PROGRAM Filtre;                    { Filtre.Pas }

USES FHandle;

VAR Taille     : LongInt;
    Buf, Ch    : Tab;
    Motif      : String;
    Nb, Iter   : Word;

{ Dans FHandle, utilise :                                   }
{  Function LitFichierLogique(h:Handle; Nb:Word;            }
{                             Var Buf:Tab) : Word;          }
{  Function EcritFichierLogique(h:Handle; Nb:Word;          }
{                               Var Buf:Tab):Word;          }
{  Function FSeek(Methode:Byte; h:Handle;                   }
{                 Position:LongInt):LongInt;                }
{ les constantes :                                          }
{   InputDos, OutputDos                                     }
{ les Types :                                               }
{   Handle, StrAsciiZ, et Tab                               }
{ et les variables :                                        }
{   Erreur, ErreurDos                                       }

 FUNCTION InverseCaps(VAR S : String) : String;
 VAR SM : String;
     i  : Byte;
 BEGIN
   SM := S; i := 1;
   WHILE (i <= Length(S)) DO
   BEGIN
     IF (S[i] IN ['A'..'Z']) THEN
       SM[i] := Chr(Ord(S[i]) + 32)
     ELSE
       SM[i] := Upcase(S[i]); Inc(i);
   END;
   InverseCaps := SM;
 END;

 FUNCTION Cherche(VAR Buf : Tab; VAR Motif : String;
                  Debut, Max : LongInt) : Word;
 VAR Maj, S : String;
     i, k   : LongInt;
     j      : Byte;
```

➥

Programme Filtre.Pas (suite). ❷

```
      NbOk    : Word;
      Ok      : Boolean;
      AdrS    : Pointer;
  BEGIN
    Maj := InverseCaps(Motif); i := Debut;
    Ok := False; j := 1; NbOk := 0;
    WHILE (i <= (Max - Length(Motif))) DO
      IF ((Mem[Seg(Buf^):Ofs(Buf^) + i] = Ord(Motif[j])) OR
      (Mem[Seg(Buf^):Ofs(Buf^) + i] = Ord(Maj[j]))) THEN
      BEGIN
        k := Length(Motif) + i - 1; j := Length(Motif);
        Ok := True;
        WHILE ((k >= i) AND Ok) DO
        IF ((Mem[Seg(Buf^):Ofs(Buf^)+k] <> Ord(Motif[j]))
           AND (Mem[Seg(Buf^):Ofs(Buf^) + k] <>
                Ord(Maj[j]))) THEN
          Ok := False
        ELSE
        BEGIN
          Dec(k); Dec(j);
        END;
        IF Ok THEN
        BEGIN
          Str(k, S);
          S := #13#10 + ' Chaîne "' + Motif +
          '" trouvée à l''offset ' + S + #13 + #10;
          S[Length(S) + 1] := #0;
          AdrS := Ptr(Seg(S), Ofs(S) + 1);
          Nb := EcritFichierLogique(OutputDos, Length(S),
                                    AdrS);
          Ok := False; Inc(NbOk);
        END;
        Inc(i, Length(Motif) - 1); j := 1;
      END
      ELSE
        Inc(i);
    Cherche := NbOk;
  END;

BEGIN
  IF (ParamCount < 1) THEN
  BEGIN
    Write(' Format : Cherche Motif ');
    Halt(1);
  END;
```

➥

Programme Filtre.Pas (suite). ❸

```
Motif := ParamStr(1);
WriteLn('Chargement et recherche en cours...');
Taille := FSeek(2, InputDos, 0);
Debut := FSeek(0, InputDos, 0);
 IF ((Taille > 0) AND (Erreur = 0)) THEN
BEGIN
  GetMem(Buf, Taille);
  Nb := Word(Taille);
  Nb := LitFichierLogique(InputDos, Nb, Buf);
  Iter := Cherche(Buf, Motif, Debut, Nb); WriteLn;
  WriteLn('Motif "', Motif, '" trouvé ', Iter, ' fois ');
END
ELSE
IF (Erreur = 0) THEN
BEGIN
  GetMem(Buf, 1024); Nb := 1;
  GetMem(Ch, 1);
  FillChar(Mem[Seg(Buf^):Ofs(Buf^)], 1024, 32);
  WHILE (Mem[Seg(Buf^):Ofs(Buf^) + Nb] <> 13) DO
  BEGIN
    Nb := Nb + LitFichierLogique(InputDos, 1, Ch);
    Mem[Seg(Buf^):Ofs(Buf^) + Nb]:=
       Mem[Seg(Ch^):Ofs(Ch^)];
  END;
  Iter := Cherche(Buf, Motif, 0, Nb); Taille := Nb;
  FreeMem(Ch, 1);
END
ELSE
BEGIN
  WriteLn(' Abandon : Erreur DOS n° ', Erreur);
  FreeMem(Buf, Taille);
END.
```

Conclusion

Ce chapitre devrait maintenant avoir rempli ses objectifs : les handles, la File Handle Table, la System File Table et les filtres n'ont plus de secrets pour vous. Il ne reste plus qu'à s'intéresser aux fichiers .EXE et nous aurons fait le tour des diverses mémoires du PC.

Chapitre 9

Fichiers .EXE

L'utilisateur et le programmeur sont tous deux concernés par les fichiers .EXE. L'utilisateur les lance, le programmeur les crée. Mais si les fichiers .EXE sont simples à exploiter, ils sont complexes à comprendre. Cela tient à leur nature hybride : ce sont à la fois des fichiers au même titre que les fichiers de données et des programmes. Autrement dit, ils sont à la fois statiques et dynamiques. On ne peut donc pas se contenter de décrire leur format d'enregistrement sur le disque. Il faut aussi expliquer leur comportement en mémoire et leurs relations avec les MCB, le PSP et le chargeur du DOS. Nous avons déjà examiné – plus ou moins complètement – chacun de ces éléments séparément. Il s'agit maintenant de les mettre en relation.

Pour cela, nous allons adopter une vision chronologique des événements. Nous verrons donc successivement :

1. le rôle exact de l'assembleur dans la création d'un fichier .EXE ;
2. l'intérêt de l'éditeur de liens ;
3. le format du fichier .EXE résultant de ces deux passes successives ;
4. les étapes du chargement en mémoire et les problèmes posés par l'allocation mémoire, la création du PSP, le relogement des adresses de segment relatives et l'exécution du processus.

Phases de création d'un fichier .EXE

Après l'écriture du code source, un programme est soit compilé, soit assemblé. Dans l'un et l'autre cas, il en résulte des fichiers .OBJ (sauf en Turbo Pascal, où le programme est directement traduit en exécutable). Ce n'est qu'ensuite que l'édition de liens permet la création d'un fichier .EXE. Ces deux phases (assemblage / compilation, et édition de liens) donnent lieu à la création du programme *en tant que fichier*. Nous allons voir ici le rôle de ces deux moments principaux de la création du fichier .EXE.

Fonction de l'assembleur

Les termes assembleur et compilateur sont considérés ici comme des synonymes : le seul rôle du compilateur qui nous intéresse étant celui que remplit l'assembleur, c'est-à-dire la création d'un code objet à partir des instructions d'un langage.

Le code source d'un programme contient toujours au moins une référence à une adresse interne à ce programme : c'est le point de départ du programme. Lorsqu'on écrit le programme Rien.Asm (voir le *listing 9.1*), lequel ne fait que rendre la main au DOS, on génère une référence à l'adresse de début du programme.

Listing 9.1
Programme Rien.Asm.

```
.Model Tiny             ; pas de pile
.Code                   ; début du segment de code
Org 100h
Debut:
  Mov Ax, 4C00h         ; code de fin de programme
  Int 21h               ; terminer programme
End  Debut              ; commencer à l'adresse «Debut»
```

Cette adresse est celle qui donnera sa valeur initiale au pointeur de programme IP. Une autre adresse, en relation avec IP, est cependant nécessaire au fonctionnement de ce programme : celle du segment de code. Or, comme pour tout programme exécutable, on ne la connaîtra qu'au moment de l'exécution. Dans ces conditions, que peut faire l'assembleur ? Il lui faut à la fois indiquer ces deux adresses dans le fichier .OBJ, mais il ne peut en connaître qu'une avec précision. Le segment de code sera par conséquent indiqué dans le fichier objet comme ayant la valeur fictive 0000h. Ce qui signifie que le code objet d'un programme contient des adresses de segment relatives et des adresses d'offset absolues. Lorsqu'un fichier contient de nombreuses procédures, dont certaines sont lointaines, le segment de code indiqué est relatif à 0000h. Ainsi, le programme Pascal de la *figure 9.2*, pourtant simple, donne le code désassemblé de la même figure.

```
PROGRAM Rien;        { Le programme Pascal }
BEGIN
END.

CS:0130    Call  0002:0000  ; Le même, tel qu'il est
CS:0135    Push  Bp         ; stocké sur le disque
CS:0136    Mov   Bp, Sp
CS:0138    Mov   Sp, Bp
CS:013A    Pop   Bp
CS:013B    Xor   Ax, Ax
CS:013D    Call  0002:00D8
                            ; Si CS = 552C
552C:0000    Call  552E:0000   ; Le même une fois en
552C:0005    Push  Bp          ; mémoire au segment
552C:0006    Mov   Bp, Sp      ; 552Ch
552C:0008    Mov   Sp, Bp
552C:000A    Pop   Bp
552C:000B    Xor   Ax, Ax
552C:000D    Call  552E:00D8
```

Encadré 9.2
Programme Rien.Pas.

Les procédures correspondant aux mots-clés BEGIN et END, qui sont chargées d'effectuer différentes initialisations, sont situées dans des segments lointains. On les appelle à l'aide de CALL FAR : d'où l'adresse de segment relative 0002h, qui se traduit bien lorsque le programme est chargé en mémoire par un appel deux segments après le segment de code en cours. Les déplacements auxquels se trouvent les procédures sont les mêmes, que le programme soit stocké sur disque ou en mémoire.

C'est le programme assembleur – ou compilateur – qui se charge de ces adressages relatifs. Ceux-ci sont stockés dans le fichier objet. Comme plusieurs fichiers objets peuvent être liés en un seul programme exécutable, des conflits sont inévitables.

Fonction de l'éditeur de liens

C'est l'éditeur de liens (LINK du DOS, ou TLINK du Turbo Assembleur) qui est chargé de résoudre les éventuels conflits entre adresses.

Lorsque deux fichiers objets sont liés dans le même programme exécutable, chacun possède en effet son adresse de début, son adresse de fin, son adresse de segment de données et son adresse de segment de pile. L'éditeur de liens lit les fichiers .OBJ et réunit les adresses de même type dans le même segment s'il le peut. Sinon, il leur donne à toutes au moins la même base et ajuste les adresses relatives de chaque fichier de telle façon que le programme .EXE soit cohérent. Si chacun des fichiers contient une adresse relative 0002h correspondant à une procédure, le fichier final contiendra une adresse relative par procédure. Si c'est l'adresse d'offset qui se trouve être la même, l'une des deux sera modifiée, la règle étant qu'à un objet (procédure, sous-programme, données, etc.) correspond une seule adresse. Le schéma qui suit (*figure 9.3*) illustre ce principe.

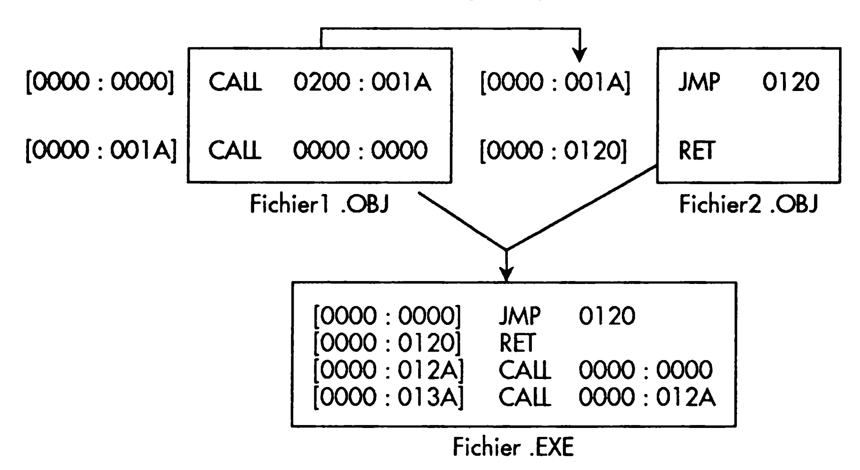


Figure 9.3
Deux fichiers .OBJ, un seul programme .EXE.

Le fichier .EXE une fois créé, il contient donc plusieurs adresses relatives (CS, SS et les adresses de segment des diverses procédures), des adresses absolues (IP, SP, les offset des procédures et des données), et des instructions. Nous allons voir de quelle façon ces adresses sont stockées.

Format d'un fichier .EXE

Un fichier .EXE contient trois parties :

1. l'en-tête (*header*),
2. la table des relogements (*relocation table*),
3. le code, les données et la pile.

Nous allons analyser les deux premières l'une après l'autre.

En-tête

L'en-tête du fichier .EXE comporte plusieurs renseignements essentiels lors du chargement en mémoire. Il a une taille fixe de 27 (1Bh) octets. Son format est donné au *tableau 9.4*.

Offset	*Description*
0000h	Mot de signature : 4D5Ah
0002h	Longueur du fichier MOD 512
0004h	Longueur du fichier DIV 512
0006h	Nombre de relogements
0008h	Taille de l'en-tête + taille de la table de relogement
000Ah	MinAlloc
000Ch	MaxAlloc
000Eh	Offset du segment de pile (SS)
0010h	Valeur de SP en début de programme
0012h	Checksum
0014h	Valeur de IP en début de programme
0016h	Offset du segment de code (CS)
0018h	Offset auquel se trouve le premier relogement dans la table de relogement
001Ah	Numéro d'overlay (normalement 0)

Tableau 9.4
Format de l'en-tête d'un fichier .EXE.

Le mot de signature permet de vérifier que le fichier contient bien un programme .EXE. La longueur totale du fichier s'obtient en additionnant le reste (en 0002h) au nombre de pages (en 0004h) multiplié par 512. Si celui-ci est supérieur à zéro, il faut retirer 512 au tout (voir la *figure 9.5*).

```
PROCEDURE LongueurFichier;
VAR LngFicDiv, LngFicMod : Word;
    LngFic               : LongInt;

BEGIN
 ...   (initialise le tableau)
 LngFic := LngFicDiv;
 LngFic := LngFic SHL 9;
 IF (LngFicMod <> 0) THEN
 BEGIN
   Dec(LngFic, 512);
   Inc(LngFic, LngFicMod);
 END;
END;
```

Encadré 9.5
Calculer la taille du fichier en octets.

Le nombre de relogements indique combien d'adresses relatives doivent être adaptées lors du chargement du programme en mémoire. Multiplié par quatre, il permet également de déterminer la taille utile de la table de relogement. Le champ qui suit est souvent intitulé «taille de l'en-tête». Comme nous dissocions ici, pour des raisons pratiques de programmation, l'en-tête de la table de relogement, cette désignation serait erronée. Il s'agit exactement de la taille prise dans le fichier par l'en-tête (1Bh octets) plus la table de relogement, dont la taille est variable. Cette taille, exprimée en paragraphes, doit être multipliée par 16 pour connaître le nombre exact d'octets pris par ces deux structures.

Attention ! le nombre ainsi obtenu a de grandes chances de ne pas être égal au nombre de relogements multiplié par quatre et additionné à 1Bh. La raison en est que le DOS arrondit le tout au paragraphe supérieur. Par conséquent, pour trouver le nombre d'octets utile pris par les relogements, il faut bien multiplier le nombre de relogements par quatre. En revanche, si l'on souhaite déterminer la taille du code et des données à charger en mémoire, on doit soustraire de la taille du fichier la valeur indiquée par le champ d'offset 0008h multipliée par 16.

```
NbOctetsReloc := NbRelogement SHL 2; { * 4  }
TailleHeader  := TailleHeader SHL 4; { * 16 }
NbOctetsFic   := LngFic - TailleHeader;
Inutilisés    := TailleHeader - (NbRelogement + $1B);
```

Encadré 9.6
Calculer la taille du code et des données.

Le champ MinAlloc (000Ah) indique le nombre de paragraphes supplémentaires minimum à allouer au programme lors de son chargement. Si la mémoire n'est pas suffisante pour allouer au moins (MinAlloc * 16) + NbOctetsFic, le programme ne sera pas chargé et l'utilisateur recevra un message d'erreur du type «mémoire insuffisante». Le champ MaxAlloc (000Ch) indique le nombre de paragraphes supplémentaires maximum à allouer au programme lors de son chargement. Le plus souvent, ce champ contient la valeur FFFFh, ce qui revient à demander toute la mémoire disponible.

Le champ 000Eh donne la valeur relative du segment de pile. Cette valeur est relative au début de l'image mémoire du fichier à charger, dont le premier segment est supposé être 0000h. L'adresse suivante (0010h) contient la valeur absolue du pointeur de pile (SP) au début du programme. Le champ suivant contient la valeur du mot de vérification du programme et ne nous intéresse pas ici. On trouve ensuite la valeur du pointeur d'instruction (IP), puis l'offset relatif auquel se trouve le segment de code (CS), qui est généralement, mais pas toujours, 0000h. Enfin, l'offset de début de la table de relogement a l'intérêt de permettre de lire la table et d'effectuer les relogements. Le numéro d'overlay, dans un programme .EXE, est le plus souvent 0. Une autre valeur permettrait de savoir quelle portion de code est en train de se charger.

Le programme Header.Pas vous permet de connaître toutes ces informations au sujet de n'importe quel fichier .EXE. Il lit, interprète et affiche l'en-tête d'un fichier dont on lui a passé le nom complet (chemin d'accès compris) en paramètre.

Listing 9.7
Programme Header.Pas. ❶

```
PROGRAM LitEnTeteExe;                { Header.Pas }

USES Crt, FHandle, Sys;

TYPE
 EnTeteExe = RECORD
               Signature1,          { 00h }
               Signature2 : Byte;  { 01h }
               TailleMod,           { 02h }
               TailleDiv,           { 04h }
               NbReloc,             { 06h }
               TailleEnTete,        { 08h }
               MinAlloc,            { 0Ah }
               MaxAlloc,            { 0Ch }
               OfsSS,               { 0Eh }
               AbsSP,               { 10h }
               CheckSum,            { 12h }
               AbsIP,               { 14h }
               OfsCS,               { 16h }
```

➥

Programme Header.Pas (suite). ❷

```
                OfsReloc,          { 18h }
                NoOvl      : Word; { 1Ah }
              END;         { Record }

VAR EnTete     : EnTeteExe;
   Buf        : Tab;
   S          : StrAsciiZ;
   Param      : String;
   i, Sauve   : Byte;

 FUNCTION LitFichierExe(VAR Nom : StrAsciiZ) : BOOLEAN;
 VAR h           : Handle;
     NbLus, Nb : Word;
 BEGIN
   Nb := $20;
   GetMem(Buf, Nb);
   h := OuvreFichier(2, Nom);
   NbLus := LitFichierLogique(h, Nb, Buf);
   IF ((Mem[Seg(Buf^):Ofs(Buf^)] = $4D) AND
   (Mem[Seg(Buf^):Ofs(Buf^) + 1] = $5A)) THEN
     LitFichierExe := True
   ELSE
   BEGIN
     FreeMem(Buf, Nb);
     LitFichierExe := False;
   END;
   FermeFichier(h);
 END;

 PROCEDURE CopieEnTete;
 BEGIN
   WITH EnTete DO
   BEGIN
     Signature1  := Mem[Seg(Buf^):Ofs(Buf^)];
     Signature2  := Mem[Seg(Buf^):Ofs(Buf^) + 1];
     TailleMod   := MemW[Seg(Buf^):Ofs(Buf^) + 2];
     TailleDiv   := MemW[Seg(Buf^):Ofs(Buf^) + 4];
     NbReloc     := MemW[Seg(Buf^):Ofs(Buf^) + 6];
     TailleEnTete := MemW[Seg(Buf^):Ofs(Buf^) + 8];
     MinAlloc    := MemW[Seg(Buf^):Ofs(Buf^) + $A];
     MaxAlloc    := MemW[Seg(Buf^):Ofs(Buf^) + $C];
     OfsSS       := MemW[Seg(Buf^):Ofs(Buf^) + $E];
     AbsSP       := MemW[Seg(Buf^):Ofs(Buf^) + $10];
     CheckSum    := MemW[Seg(Buf^):Ofs(Buf^) + $12];
```

➥

Programme Header.Pas (suite). ❸

```
    AbsIP       := MemW[Seg(Buf^):Ofs(Buf^) + $14];
    OfsCS       := MemW[Seg(Buf^):Ofs(Buf^) + $16];
    OfsReloc    := MemW[Seg(Buf^):Ofs(Buf^) + $18];
    NoOvl       := MemW[Seg(Buf^):Ofs(Buf^) + $1A];
  END;
END;

PROCEDURE AfficheEnTete;
VAR TailleFic, Max, Min : LongInt;
BEGIN
  TextAttr := 14 + 4 * 16;
  GotoXy(5, 5);
  Write(' Signature                          : ');
  GotoXy(5, 6);
  Write(' Taille DIV 512                     : ');
  GotoXy(5, 7);
  Write(' Taille MOD 512                     : ');
  GotoXy(5, 8);
  Write(' Nombre de relogements              : ');
  GotoXy(5, 9);
  Write(' Taille de l''en-tête en paragraphes : ');
  GotoXy(5, 10);
  Write(' Minimum à allouer en plus du fichier : ');
  GotoXy(5, 11);
  Write(' Maximum à allouer en plus du fichier : ');
  GotoXy(5, 12);
  Write(' SS à l''Offset                      : ');
  GotoXy(5, 13);
  Write(' SP =                               : ');
  GotoXy(5, 14);
  Write(' Checksum                           : ');
  GotoXy(5, 15);
  Write(' IP =                               : ');
  GotoXy(5, 16);
  Write(' CS à l''Offset                      : ');
  GotoXy(5, 17);
  Write(' Premier relogement en              : ');
  GotoXy(5, 18);
  Write(' Numéro d''Overlay                   : ');
  TextAttr := 15 + 3 * 16;
  WITH EnTete DO
  BEGIN
    GotoXy(45, 5);
    Write(' ', OctetDecVersHex(Signature2),
               OctetDecVersHex(Signature1), 'h', '':18);
```

➥

Programme Header.Pas (suite). ❹

```
      GotoXy(45, 6);
      Write(' ', MotDecVersHex(TailleDiv), 'h', '':18);
      GotoXy(45, 7);
      Write(' ', MotDecVersHex(TailleMod), 'h', '':18);
      GotoXy(45, 8);
      Write(' ', NbReloc:4, ' ', '':18);
      GotoXy(45, 9);
      Write(' ', MotDecVersHex(TailleEnTete), 'h', '':18);
      GotoXy(45, 10);
      Write(' ', MotDecVersHex(MinAlloc), 'h', '':18);
      GotoXy(45, 11);
      Write(' ', MotDecVersHex(MaxAlloc), 'h', '':18);
      GotoXy(45, 12);
      Write(' ', MotDecVersHex(OfsSS), 'h', '':18);
      GotoXy(45, 13);
      Write(' ', MotDecVersHex(AbsSP), 'h', '':18);
      GotoXy(45, 14);
      Write(' ', MotDecVersHex(CheckSum), 'h', '':18);
      GotoXy(45, 15);
      Write(' ', MotDecVersHex(AbsIP), 'h', '':18);
      GotoXy(45, 16);
      Write(' ', MotDecVersHex(OfsCS), 'h', '':18);
      GotoXy(45, 17);
      Write(' ', MotDecVersHex(OfsReloc), 'h', '':18);
      GotoXy(45, 18);
      Write(' ', NoOvl:4, ' ', '':18);
      TailleFic := TailleDiv; TailleFic := TailleFic SHL 9;
      Inc(TailleFic, TailleMod);
      Min := MinAlloc; Min := Min SHL 4;
      Max := MaxAlloc; Max := Max SHL 4;
      GotoXy(50, 6); Write(TailleFic:10, ' octets ');
      GotoXy(50, 9);
      Write((TailleEnTete * 16):10, ' octets ');
      GotoXy(50, 10); Write(Min:10, ' octets ');
      GotoXy(50, 11); Write(Max:10, ' octets ');
    END;
  END;

BEGIN
   IF (ParamCount < 1) THEN
   BEGIN
     WriteLn(' Nom du fichier .EXE ! '); Halt(1);
   END;
```

➥

Programme Header.Pas (suite). ❺

```
    Sauve := TextAttr;
    Param := ParamStr(1); FillChar(S, SizeOf(S), 0);
    Move(Param[1], S[0], Length(Param));
    i := 0;
    WHILE (S[i] > #0) DO
      Inc(i);
    S[i] := #0;
    IF LitFichierExe(S) THEN
    BEGIN
      CopieEnTete;
      AfficheEnTete;
      ReadLn;
    END;
    TextAttr := Sauve; ClrScr;
END.
```

Après l'en-tête, mais inutilisable sans lui, vient la table des relogements, qui permet de charger correctement le fichier en mémoire.

Table des relogements

Nous l'avons dit, un fichier `.EXE` contient des adresses de segment, qui correspondent le plus souvent à des procédures lointaines (FAR) ou proches (NEAR). Ces adresses sont inutilisables en tant que telles, parce qu'elles sont nécessairement relatives, alors qu'elles devront être absolues une fois le programme en mémoire. Il fallait donc une structure chargée de les localiser à l'intérieur du fichier. La table des relogements tient ce rôle.

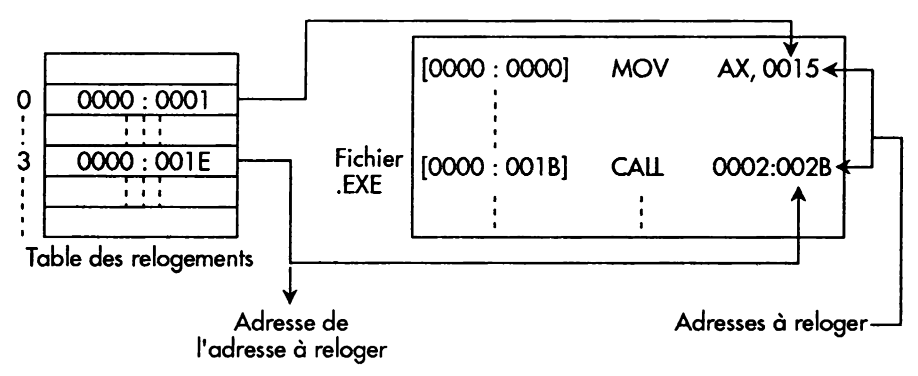


Figure 9.8
La table des relogements : format et fonctionnement.

Chaque adresse à reloger est identifiée dans cette table par une autre adresse, donnée sous la forme `segment:déplacement`. Lorsqu'on se déplace dans le fichier à l'adresse indiquée par la table des relogements, on trouve l'adresse à

reloger. La table des relogements est donc constituée d'un nombre variable d'entrées. Chaque entrée, sur deux mots, contient l'adresse d'offset et l'adresse de segment (relative) où trouver l'adresse à reloger dans le fichier inscrit sur le disque. Le fonctionnement est schématisé à la *figure 9.9.*

Si nous compilons le programme RIEN.PAS, le format de son en-tête et celui de sa table de relogement permettent de localiser les adresses à reloger. C'est ce que montre la *figure 9.9.*

```
-n rien.bin
-l
-d 0100 l 2  Cs:0100  4D 5A  ; Signature
-d 0102 l 2  Cs:0100  20 01  ; Taille MOD 512 =
-d 0104 l 2  Cs:0100  03 00  ; Taille DIV 512 = 3
-d 0106 l 2  Cs:0100  04 00  ; 4 relogements
-d 0108 l 2  Cs:0100  03 00  ; Header = 3*16 octets
-d 010a l 2  Cs:0100  25 04  ; MinAlloc = 0425h * 16
-d 010c l 2  Cs:0100  25 A4  ; MaxAlloc = A425h * 16
-d 010e l 2  Cs:0100  74 00  ; SS en 0074h
-d 0110 l 2  Cs:0110  00 40  ; SP = 0040h
-d 0112 l 2  Cs:0110  00 00  ; Checksum = 0
-d 0114 l 2  Cs:0110  00 00  ; IP = 0000
-d 0116 l 2  Cs:0110  00 00  ; CS en 0000
-d 0118 l 2  Cs:0110  1C 00  ; table en 001Ch
-d 011a l 2  Cs:0110  00 00  ; Numéro d'overlay = 0

-d 011C l 4  011C  03 00 00 00 ;Adresse n° 1 en 0000:0003
-d 0120 l 4  0120  10 00 00 00 ;Adresse n° 2 en 0000:0010
-d 0124 l 4  0124  01 00 02 00 ;Adresse n° 3 en 0002:0001
-d 0128 l 4  0128  DD 00 02 00 ;Adresse n° 4 en 0002:00DD

-u 0130 013d
        0 1 2 3 4
Cs:0130 9A00000200    CALL  0002:0000
              └──────────────┘
Cs:0135 55            PUSH   BP
Cs:0136 89E5          MOV    BP,SP
Cs:0138 89EC          MOV    SP,BP
Cs:013A 5D            POP    BP
Cs:013B 31C0          XOR    AX,AX
        D E F 10 11
Cs:013D 9AD8000200    CALL   0002:00D8
              └───────────────┘
-u 1743:0130
          0 1 2
1743:0130 BA4B00      MOV  DX,004B
...          └──────────────────┘
...       C D E
1743:020C BA4B00      MOV  DX,004B
...          └──────────────────┘
```

Figure 9.9
En-tête, table de relogement et code de Rien.EXE.

Rien.EXE une fois renommé en .BIN, DEBUG ne peut plus reconnaître le format et permet ainsi d'examiner le code brut, avec adresses relatives etc. Le code commence en Cs:0130h, en 0100h se trouve l'en-tête. La table de relogement commence en 010Ch et contient quatre octets inutilisés. En 0133h (0130h + 0003h) se trouve la première adresse à reloger. En 0140h (0130h + 0010h) se trouve la seconde. Les troisième et quatrième adresses sont deux segments plus loin (CS a la valeur 1741h).

Les informations qui se trouvent dans l'en-tête et dans la table de relogement servent durant la phase de chargement en mémoire. C'est cette étape que nous allons maintenant aborder.

Chargement d'un fichier .EXE

L'une des opérations les plus complexes que le DOS ait à effectuer est bien celle du chargement d'un programme .EXE en mémoire. Une fois le fichier localisé sur disque, le DOS doit :

1. réduire sa propre taille en mémoire,
2. lire l'en-tête de fichier .EXE,
3. déterminer les besoins en mémoire du programme,
4. allouer suffisamment de mémoire pour le programme et son PSP,
5. créer le PSP en mémoire,
6. lire le fichier .EXE et le charger,
7. lire la table de relogement du fichier,
8. procéder à la mise à jour des adresses mémoires dans le fichier,
9. lancer le programme.

Chacune de ces opérations suppose des vérifications. Nous allons toutes les voir dans le détail.

Réduction de la mémoire

Si c'est bien COMMAND.COM qui lance le programme, cette étape ne pose pas de difficulté particulière. On sait en effet que l'interpréteur de commandes du DOS est chargé en deux parties : partie résidente en mémoire basse, partie transitoire en mémoire haute. La partie résidente ne se déchargeant pas de la mémoire, il suffit de lui rendre le contrôle pour que la partie transitoire libère la partie de la mémoire qu'elle occupe. En d'autres termes, elle peut être écrasée impunément par le nouveau programme.

En revanche, si l'on lance un programme fils à partir d'un autre, le père doit déterminer la place qu'il occupe et celle dont il a réellement besoin. Une fois qu'il a réussi à calculer ses propres besoins, il peut faire appel à la fonction `4Ah` de l'Int `21h` du DOS pour libérer la mémoire dont il n'a pas besoin. La difficulté réside naturellement dans le calcul exact de ses propres besoins. Deux méthodes sont envisageables : soit le programme père soustrait le segment de son PSP du segment de pile et ajoute au tout la valeur maximale que peut prendre le pointeur de pile, soit il a lu au lancement la valeur `MinAlloc` dans son propre fichier et connaît donc le minimum vital dont il a besoin en plus de la taille du fichier pour fonctionner correctement.

Listing 9.10
Programme VirMem.Pas. ❶

```
PROGRAM JouerAvecLaMemoire;          { VirMem.Pas }

{$M 16384, 0, 65536}

USES Dos;

VAR  SegPrg, Resultat,
     Memoire, PSP,
     PointeurPile      : Word;
     NbMaxParag        : LongInt;

FUNCTION AlloueMemoire(NbPara : Word;
                       VAR Segment : Word) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $48;
    Bx := NbPara;
    MsDos(Regs);
    Segment := Ax;
    IF (Flags AND 1 = 1) THEN
      AlloueMemoire := Bx       { Maximum Disponible }
    ELSE
      AlloueMemoire := 0;       { Tout s'est bien passé }
  END;
END;

FUNCTION ModifieMemoire(NbPara, Segment : Word) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
```

➥

Programme VirMem.Pas (suite). ❷

```
  BEGIN
    Ah := $4A;
    Bx := NbPara;
    Es := Segment;
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
      ModifieMemoire := Bx          { Maximum disponible }
    ELSE
      ModifieMemoire := 0;          { Tout s'est bien passé }
  END;
END;

FUNCTION TailleMaxPrg(PSP, PointeurPile : Word) : Word;
VAR NbSegments : Word;
BEGIN
  NbSegments := (SSeg - PSP);
  NbSegments := (NbSegments + PointeurPile + 1) SHR 4;
  TailleMaxPrg := NbSegments;
END;

BEGIN
  PSP := PrefixSeg; PointeurPile := SPtr;
  Memoire := TailleMaxPrg(PSP, PointeurPile);

  SegPrg := PSP; Resultat := AlloueMemoire($FFFF, SegPrg);
  NbMaxParag := Resultat; NbMaxParag := NbMaxParag SHL 4;
  WriteLn(' Bloc Maximum Disponible : ',
          NbMaxParag SHR 10, ' Ko');

  Resultat := ModifieMemoire(Memoire, PSP);
  WriteLn(Resultat);

  Resultat := AlloueMemoire($FFFF, SegPrg);
  NbMaxParag := Resultat; NbMaxParag := NbMaxParag SHL 4;
  WriteLn(' Bloc Maximum Disponible : ',
          NbMaxParag SHR 10, ' Ko');
END.
```

Si l'opération de réduction de la mémoire disponible échoue et que le programme père occupe toute la mémoire disponible – comme c'est généralement le cas – on ne peut charger de programme supplémentaire en mémoire. Sinon, l'opération peut se poursuivre.

Lire l'en-tête de fichier

Cette seconde étape nécessite plusieurs conditions. Tout d'abord, il faut disposer dans sa zone de données d'un bloc mémoire destiné à recueillir les renseignements contenus par l'en-tête. Cela semble sans doute une petite contrainte, mais il ne faut pas oublier que la mémoire vive est comptée. Ensuite, il faut ouvrir le fichier : c'est un premier test. Après quoi, on fait une lecture bufferisée (fonction de lecture de fichier ou périphérique) du fichier. Il y a, là aussi, un test à effectuer. Ensuite les opérations se déroulent de la façon suivante :

1. vérification de la signature du fichier. Si ce n'est pas la bonne, on abandonne tout, ou on lance une opération de chargement de fichier `.COM`
2. vérification de la valeur de checksum. Si le fichier est endommagé, on arrête l'opération
3. lecture de la taille de fichier, calcul de l'image mémoire et de la taille de la table de relogement
4. mémorisation des autres données dans des variables.

Déterminer les besoins en mémoire

A partir de la taille du code et des champs `MinAlloc` et `MaxAlloc`, il est relativement simple de calculer les besoins mémoire. Le programme père ajoute la valeur de `MaxAlloc` à la taille de l'image mémoire du fichier. Si le total est inférieur ou égal à la quantité de mémoire disponible, il passe à l'étape suivante. Sinon, si la quantité de mémoire disponible est supérieure à la taille du fichier plus `MinAlloc` et inférieure à la taille du fichier plus `MaxAlloc`, le programme père allouera toute la mémoire disponible. Sinon, si la taille mémoire est inférieure à la taille du fichier plus `MinAlloc`, le programme père interrompt le processus de chargement.

```
FUNCTION DetermineBesoinsMemoire : WORD;
VAR Regs        : Registers;
    Necessaire,
    TailleMax   : Word;
BEGIN
 WITH Regs DO
 BEGIN
   Ah := $48;
   Bx := $FFFF; { Requête impossible à satisfaire }
   MsDos(Regs); { Mémoire disponible renvoyée dans BX }
   TailleMax := Bx; { en paragraphes }
 END;
 Necessaire := TailleFic + MaxAlloc + $100;
```

```
  IF (TailleMax >= Necessaire) THEN
    DetermineBesoinsMemoire := Necessaire
  ELSE
  BEGIN
    Necessaire := TailleFic + MinAlloc + $100;
    IF (TailleMax >= Necessaire) THEN
      DetermineBesoinsMemoire := Necessaire
    ELSE
      DetermineBesoinsMemoire := 0;
  END;
END;
```

Encadré 9.11
Déterminer les besoins en mémoire d'un programme fils.

Comme la pseudo-fonction de l'encadré le montre, `100h` (256) octets sont ajoutés à la taille mémoire nécessaire au chargement du fichier : ils sont destinés au PSP.

Allocation mémoire

Une fois les besoins du fils connus, le programme père alloue la mémoire nécessaire à son fonctionnement par l'intermédiaire de la fonction `48h` de l'Int `21h` du DOS. Le système crée alors un MCB (*memory control block*) supplémentaire en mémoire et renvoie à son père l'adresse de segment sur laquelle il pointe. La structure des MCB a été étudiée au chapitre 3 (*RAM gérée par le DOS*). Le seul ennui qui peut alors arriver est que la RAM ne puisse pas contenir à la fois le MCB et le segment pointé. Comme le MCB ne fait que `10h` (16) octets, c'est particulièrement rare. Dans un tel cas, le programme père peut tenter de réallouer la mémoire moins 16 octets plutôt que de tout arrêter.

Créer le PSP

Le DOS connaît maintenant l'adresse de segment où charger le programme. Comme il a également en mémoire les éventuels arguments passés par l'utilisateur, il peut créer le PSP sans grande difficulté.

En premier lieu, le DOS crée le PSP en faisant appel à la fonction `55h` de l'Int `21h` (non documentée). Il passe l'adresse de segment en `Dx` et le numéro de fonction en `Ah`. La majeure partie du PSP est alors initialisée.

Description	Entrées	Sorties
Crée un PSP, initalise tous les champs, sauf la ligne de commande et les FCB	AH := 55h DX := Seg(PSP)	Rien
Crée un FCB à partir d'un nom de fichier présent dans une ligne de commande	AH := 29h AL := Mode d'analyse DS := Seg(chaîne) SI = Ofs(chaîne) ES = Seg(FCB) DI := Ofs(FCB)	AL = joker DS = Seg(nom) SI := Ofs(nom) ES := Seg(FCB) DI = Ofs(FCB)
Installe le PSP courant	AH := 50h BX := Seg(PSP)	Rien

Format du mode d'analyse (fonction 29h) :

Bit	Valeur	Description
3	1	Si une extension est présente dans la chaîne, le champ d'extension sera modifié. Si la chaîne ne précise pas d'extension, le champ du FCB sera rempli par des espaces.
2	1	Si un nom de fichier est présent dans la chaîne, le champ du FCB sera modifié. Si la chaîne ne précise pas de nom de fichier, le champ du FCB sera rempli par des espaces.
1	1	Si la chaîne précise un média, le champ correspondant dans le FCB sera modifié.
	0	Si la chaîne ne précise pas de média, le champ sera rempli par des espaces.
0	1	Les séparateurs de la chaînes sont ignorés.
	0	Les séparateurs ne sont pas examinés.

Format de l'octet "Joker" :

Valeur	Signification
00h	Pas de caractère joker
01h	Il y a des caractères jokers
FFh	Identificateur de média invalide

Tableau 9.12
Fonctions utilisées par le DOS pour créer le PSP.

Il reste toutefois à recopier la ligne de commande dans le champ correspondant du PSP et à installer les deux champs FCB. Pour remplir cette dernière tâche, le DOS utilise la fonction `29h` de l'Int `21h` et lui passe la valeur `01h` dans l'octet de mode d'analyse.

Le PSP est alors complètement initialisé, mais ne sera pas installé en tant que PSP courant avant le lancement du programme.

Lire le fichier .EXE et le charger

Le chargeur du DOS déplace ensuite le pointeur de fichier vers l'adresse de début du code par l'intermédiaire de la fonction `42h` de l'Int `21h`. Il lit le code, les données et la pile du programme dans la zone mémoire allouée qui se trouve juste au dessus du PSP, c'est-à-dire 256 octets plus loin. Pour trouver l'adresse, on ajoute `10h` (`16`) au numéro de segment du PSP. Si le déplacement de pointeur ou la lecture échoue, le DOS recommence l'opération plusieurs fois de suite avant d'abandonner.

Lire la table de relogement

Le DOS déplace ensuite le pointeur de fichier vers l'offset de la première adresse de relogement et charge une partie des adresses dans un buffer interne. La table de relogement n'étant pas limitée en taille, le buffer fait sans doute un maximum de 2 Ko (512 adresses * 4 octets). De toutes les façons, il faut passer à l'opération suivante pour ensuite boucler jusqu'à la fin du traitement des adresses de relogement.

Si le buffer était créé dynamiquement en RAM, sa taille ne serait pas limitée. Mais sa création interviendrait alors forcément avant que le DOS n'alloue de la mémoire au fichier `.EXE`, ce qui poserait deux problèmes. D'une part, les chances d'exécution du programme s'en trouveraient réduites, puisqu'il disposerait de moins de mémoire. D'autre part, on serait obligé de supprimer le buffer de la mémoire entre la fin de l'opération de relogement et le début de l'exécution du programme. Cela fragmènterait la mémoire et gênerait considérablement le chargement de programmes importants. Il est donc impossible que le buffer soit alloué dynamiquement.

Reloger les adresses

Le relogement des adresses que contient le fichier `.EXE` est une phase cruciale du chargement. Aucune erreur ne doit survenir, sans quoi il serait impossible d'exécuter correctement le programme fils.

Chaque entrée de la table de relogement pointe sur l'emplacement du fichier où se trouve l'adresse à reloger. Une fois cette adresse trouvée dans le fichier, on lui ajoute la valeur relative de CS indiquée par l'en-tête de fichier .EXE et le numéro de segment où se trouve effectivement le code en mémoire. La valeur indiquée par l'en-tête est généralement 0000h. Le numéro de segment réel est imprévisible. On l'a identifié lors du chargement de l'image mémoire au dessus du PSP.

Le programme qui suit, Reloc.Pas, charge un programme .EXE en mémoire, obtient les informations nécessaires de l'en-tête et affiche l'adresse de relogement, la valeur de l'adresse à reloger dans le fichier et la valeur qu'elle prendrait si elle était relogée. Le programme ne procède pas au relogement : il faudrait pour cela qu'il fonctionne comme le chargeur du DOS. Mais il devrait permettre de comprendre comment se déroule l'opération.

Listing 9.13
Programme Reloc.Pas. ❶

```
 PROGRAM LitAdressesDeRelogement;  { Reloc.Pas }
 {$M 16384, 0, 16384}

USES Dos, Crt, FHandle, Sys;

TYPE  Relogement = RECORD
                     Segment,
                     Offset : Word;
                   END;     { Record }

VAR  Buf, BufFic   : Tab;
     AdrReloc      : Relogement;
     Sauve         : Byte;
     CS, IP,
     TailleEnTete,
     NbReloc,
     OfsReloc,
     SegmentPrg    : Word;
     TailleImage   : LongInt;
     S             : String;
     Nom           : StrAsciiZ;

  FUNCTION TaillePrg : Word;
  VAR Max, Min : Word;
      Taille   : Word;
  BEGIN
    Min := PrefixSeg;
    Max := SSeg;
    Taille := Max - Min;
```

➥

Programme Reloc.Pas (suite). ❷

```
    Inc(Taille, (16384 SHR 4));
    Inc(Taille);
    TaillePrg := Taille;
  END;

  FUNCTION Shrink(NouvelleTaille : Word) : Word;
  VAR Regs : Registers;
  BEGIN
    WITH Regs DO
    BEGIN
      Ah := $4A;
      Bx := NouvelleTaille;
      Es := PrefixSeg;
      MsDos(Regs);
      IF (Flags AND 1 = 1) THEN
      BEGIN
        Erreur := Ax; Shrink := Bx;
      END
      ELSE
      BEGIN
        Erreur := 0; Shrink := 0;
      END;
    END;
  END;

  FUNCTION Alloue(NbOctets : Word) : Word;
  VAR Regs : Registers;
  BEGIN
    WITH Regs DO
    BEGIN
      Ah := $48;
      Bx := NbOctets;
      MsDos(Regs);
      IF (Flags AND 1 = 1) THEN
      BEGIN
        Erreur := Ax;
        Alloue := Bx;
      END
      ELSE
        Alloue := Ax;
    END;
  END;
```

➥

Programme Reloc.Pas (suite). ❸

```
PROCEDURE InitVar(VAR NomFic : StrAsciiZ);
VAR Nb, Max, NbOctets : Word;
    h                 : Handle;
    Position          : LongInt;
    Reste             : Word;
BEGIN
  Nb := $20;
  GetMem(Buf, Nb);
  h := OuvreFichier(2, NomFic);
  IF (Erreur = 0) THEN
  BEGIN
    Nb := LitFichierLogique(h, Nb, Buf);
    IF (Nb > 0) THEN
    BEGIN
      CS := MemW[Seg(Buf^):Ofs(Buf^) + $16];
      IP := MemW[Seg(Buf^):Ofs(Buf^) + $14];
      TailleImage := MemW[Seg(Buf^):Ofs(Buf^) + 4];
      Reste := MemW[Seg(Buf^):Ofs(Buf^) + 2];
      IF (Reste = 0) THEN
        TailleImage := (TailleImage SHL 9) + Reste
      ELSE
        TailleImage := ((TailleImage SHL 9)-512)+Reste;
      TailleEnTete := MemW[Seg(Buf^):Ofs(Buf^) + 8];
      TailleEnTete := TailleEnTete SHL 4;
      Dec(TailleImage, TailleEnTete);
      NbReloc := MemW[Seg(Buf^):Ofs(Buf^) + 6];
      OfsReloc := MemW[Seg(Buf^):Ofs(Buf^) + $18];
    END
    ELSE
      Write(' Erreur en lecture de fichier ');
  END
  ELSE
    Write(' Erreur en ouverture de fichier ');
  NbOctets := NbReloc SHL 2;
  Position := FSeek(0, h, 0);
  Position := FSeek(0, h, LongInt(OfsReloc));
  IF (Position = OfsReloc) THEN
  BEGIN
    Max := Alloue(NbOctets);
    Buf := Ptr(Max, 0);
    Nb := LitFichierLogique(h, NbOctets, Buf);
    Position := FSeek(0, h, LongInt(TailleEnTete));
    Nb := (TailleImage SHR 4);  { Taille en paragraphes }
    Reste := Shrink(TaillePrg);
```

➥

Programme Reloc.Pas (suite). ❹

```
      IF (Erreur = 0) THEN
      BEGIN
        Max := Alloue(Nb); SegmentPrg := Max;
        BufFic := Ptr(Max, 0);
        Nb := Nb SHL 4;           { Taille en octets }
        Nb := LitFichierLogique(h, Nb, BufFic);
        WriteLn(' Fichier lu ');
        Reste := Shrink($FFFF);
        WriteLn(' Mémoire disponible : ', (Reste SHL 4),
                ' octets');
      END
      ELSE
      BEGIN
        Write(' Pas assez de mémoire libre ');
        FreeMem(Buf, NbOctets);
      END;
    END;
    FermeFichier(h);
  END;

  PROCEDURE TrouveReloc;
  VAR i, SegPrg, SegDef : Word;
  BEGIN
    i := 0; CS := CS + SegmentPrg;
    WriteLn(NbReloc:7, ' adresses à reloger : ');
    WriteLn('CS:IP = ', MotDecVersHex(CS), ':',
                        MotDecVersHex(IP));
    NbReloc := NbReloc SHL 2;
    WHILE (i < NbReloc) DO
    BEGIN
      WITH AdrReloc DO
      BEGIN
        Offset  := MemW[Seg(Buf^):Ofs(Buf^) + i];
        Segment := MemW[Seg(Buf^):Ofs(Buf^) + 2 + i];
        SegPrg  := MemW[Seg(BufFic^)+Segment:
                        :Ofs(BufFic^)+Offset];
        SegDef  := SegPrg + CS;

        { Si l'on voulait reloger réellement l'adresse ,}
        { on ferait ici : }
        { MemW[Seg(BufFic^)+Segment:
               Ofs(BufFic^)+Offset] := SegDef;    }
```

➥

Programme Reloc.Pas (suite). ❺

```
        Inc(i, 4);
        Write('':4, MotDecVersHex(Segment), ':',
                    MotDecVersHex(Offset));
        Write(' -> ', MotDecVersHex(SegPrg));
        WriteLn(' =  ', MotDecVersHex(SegDef));
        IF (WhereY >= 24) THEN
        BEGIN
          ReadLn; ClrScr; WriteLn;
        END;
      END;
    END;
  END;

BEGIN
  IF (ParamCount >= 1) THEN
  BEGIN
    S := ParamStr(1);
    FillChar(Nom[0], SizeOf(Nom), 0);
    Move(S[1], Nom[0], Length(S));
    Sauve := TextAttr;
    ClrScr;
    InitVar(Nom);
    TrouveReloc;
    TextAttr:=Sauve;
  END
  ELSE
    WriteLn(' Fichier .EXE en ligne de commande ');
END.
```

En listant ainsi les adresses à reloger et les valeurs qu'elles peuvent prendre, `Reloc.Pas` permet de connaître l'adresse sur le disque de toutes les procédures du programme. Il ne s'agit donc pas seulement d'une démonstration, mais aussi d'un utilitaire bien pratique lorsque l'on souhaite désassembler un programme.

La *figure 9.14* montre comment s'effectue le relogement.

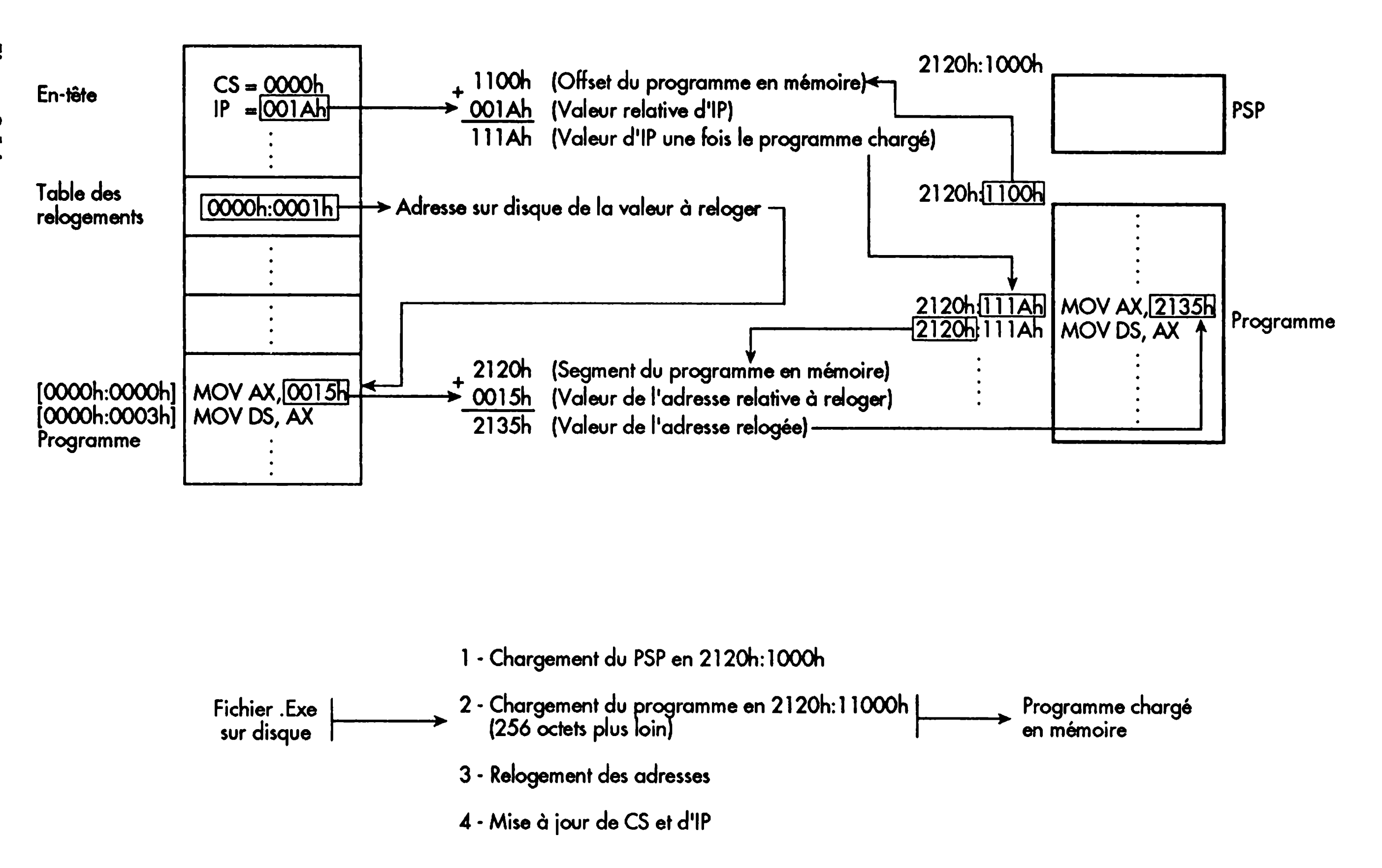


Figure 9.14
Relogement des adresses.

Lancer le programme

Pour lancer l'exécution du programme fils, le DOS commence par installer le nouveau PSP comme PSP courant. Il initialise ensuite SS et SP puis CS et IP à leurs nouvelles valeurs. A partir de ce moment tout est en ordre : les différents registres sont à jour, la mémoire est allouée, les adresses sont relogées et le PSP est installé.

Lorsque le programme .EXE s'interrompt, il rend directement la main au DOS et n'a donc pas à passer par la fonction EXEC.

Les fichiers .EXE n'ont désormais plus de secrets pour vous. Nous avons même montré comment fonctionne le chargeur du DOS suffisamment dans les détails pour que vous puissiez en programmer un. Un autre projet, simple à réaliser, serait de permettre à l'utilisateur de diminuer la valeur du champ MaxAlloc. Microsoft vend avec ses compilateurs un utilitaire de ce type – EXEMOD. Il s'agira de notre dernier programme.

Modifier MinAlloc et MaxAlloc

Il ne contient que trois procédures, faciles à comprendre. LitHeader charge le contenu de l'en-tête en mémoire et initialise les variables Min et Max, qui donnent les valeurs des champs MinAlloc et MaxAlloc. EcritHeader sauve l'en-tête sur le disque. LitClavier affiche les valeurs initiales des champs MinAlloc et MaxAlloc et permet leur modification. Un champ ne sera modifié que si l'utilisateur a entré un nombre valide. Sinon, il garde sa valeur de départ. Si MinAlloc est supérieur à MaxAlloc ou si l'on a tenté de réduire la taille de MinAlloc, la réponse est refusée et le programme affiche un message d'erreur.

Listing 9.15
Programme ModExe.Pas. ❶

```
PROGRAM ModifieMinAlloc;            { ModExe.Pas }

USES Crt, FHandle, Sys;

VAR Buf        : Tab;
    Nom        : StrAsciiZ;
    Sauve      : Byte;
    h          : Handle;
    S          : String;
    Min, Max   : Word;
```

➥

Programme ModExe.Pas (suite). ❷

```
PROCEDURE LitHeader(VAR Min, Max : Word);
VAR NbLus : Word;
BEGIN
  GetMem(Buf, $1B);
  h := OuvreFichier(2, Nom);
  NbLus := LitFichierLogique(h, $1B, Buf);
  IF (Erreur <> 0) THEN
  BEGIN
    WriteLn(' Erreur en ouverture de fichier ');
    Halt(1);
  END
  ELSE
  BEGIN
    Max := MemW[Seg(Buf^):Ofs(Buf^) + $C];
    Min := MemW[Seg(Buf^):Ofs(Buf^) + $A];
  END;
END;

PROCEDURE EcritHeader(VAR Min, Max : Word);
VAR NbEcrits : Word;
    Position : LongInt;
BEGIN
  MemW[Seg(Buf^):Ofs(Buf^) + $C] := Max;
  MemW[Seg(Buf^):Ofs(Buf^) + $A] := Min;
  Position := FSeek(0, h, 0);
  IF (Position = 0) THEN
  BEGIN
    NbEcrits := EcritFichierLogique(h, $1B, Buf);
    FreeMem(Buf, $1B);
    FermeFichier(h);
  END
  ELSE
  BEGIN
    WriteLn(' Erreur en écriture fichier ');
    Halt(1);
  END;
END;

PROCEDURE LitClavier;
VAR Min2, Max2        : Word;
    Minimum, Maximum : String;
BEGIN
  Min := 0; Max := 0;
  GotoXy(5, 3); TextAttr := 15 + 4 * 16;
```

➥

Programme ModExe.Pas (suite). ❸

```
    Write(' ', S, ' ');
    GotoXy(5, 5); Write(' MAXalloc : ');
    GotoXy(5, 8); Write(' MAXalloc : ');
    GotoXy(45, 5); Write(' MINalloc : ');
    GotoXy(45, 8); Write(' MINalloc : ');
    TextAttr := 15 + 1 * 16; GotoXy(17, 5);
    LitHeader(Min, Max);
    Write(MotDecVersHex(Max), 'h ', Max, 'd');
    GotoXy(57, 5);
    Write(MotDecVersHex(Min), 'h ', Min, 'd');
    GotoXy(17, 8); ReadLn(Maximum);
    GotoXy(57, 8); ReadLn(Minimum);
    IF ((Maximum = '') AND (Minimum = '')) THEN
    BEGIN
      FermeFichier(h);
      Halt;
    END;
    IF (Maximum = '') THEN
      Max2 := Max
    ELSE
      Max2 := ValeurNum(Maximum, 10);
    IF (Minimum = '') THEN
      Min2 := Min
    ELSE
      Min2 := ValeurNum(Minimum, 10);
    IF (Max2 <= $FFFF) THEN
    BEGIN
      GotoXy(5, 24);
      IF ((Max2 >= Min) AND (Min2 >= Min)) THEN
      BEGIN
        EcritHeader(Min2, Max2); TextAttr := 14 + 4 * 16;
        Write(' Modifications enregistrées ');
      END
      ELSE
      BEGIN
        FermeFichier(h);
        IF (Max2 < Min) THEN
          Write(' Erreur : MAXalloc ne peut être ',
                'plus petit que MINalloc ')
        ELSE
        IF (Min2 < Min) THEN
          Write(' Erreur : Ne pas allouer moins '
                'que le minimum prévu ');
      END;
```

➥

Programme ModExe.Pas (suite). ❹

```
    GotoXy(5, 24); TextAttr := 15 + 1 * 16;
    ReadLn; Write('':75);
  END;
END;

BEGIN
  IF (ParamCount >= 1) THEN
  BEGIN
    Min := 0; Max := 0;
    S := ParamStr(1);
    FillChar(Nom, SizeOf(Nom), 0);
    Move(S[1], Nom[0], Length(S));
    Sauve := TextAttr; ClrScr;
    LitHeader(Min, Max);
    LitClavier;
    TextAttr := Sauve; ClrScr;
  END
  ELSE
    WriteLn(' Nom du fichier .EXE en paramètre ');
END.
```

Conclusion

Nous avons vu tous les éléments d'un fichier .EXE les uns après les autres. Nous en avons expliqué le fonctionnement dans les détails et montré que l'opération de chargement d'un programme en mémoire était sans doute la plus complexe que le DOS ait à réaliser.

Annexe 1

Source des unités Sys et FHandle

Cette annexe présente le code source des deux unités que nous utilisons à l'intérieur de ce livre. Celles-ci contiennent de nombreuses procédures et fonctions particulièrement utiles. Nous ne saurions trop vous encourager à les étudier.

Unité Sys.Tpu

❶

```
UNIT Sys;

INTERFACE

FUNCTION At : BOOLEAN;
 { Le PC est-il un AT ou un XT ? }
FUNCTION Boole(Posit : Byte) : BOOLEAN;
 { Renvoie 'Oui' si Posit est à 1, 'Non' autrement }
FUNCTION ValeurNum(VAR Nb : STRING; Base : Byte) : Word;
 { Transforme une chaîne en chiffre }
FUNCTION OctetDecVersBin(Octet : Byte) : STRING;
 { Convertit un octet décimal en binaire }
FUNCTION MotDecVersBin(Mot : Word) : STRING;
 { Convertit un mot décimal en binaire }
FUNCTION OctetDecVersHex(Octet : Byte) : STRING;
 { Convertit un octet décimal en hexa }
FUNCTION MotDecVersHex(Mot : Word) : STRING;
 { Convertit un mot décimal en hexa }
FUNCTION HexaVersDecimal(Mot : STRING) : Word;
 { Convertit un mot ou un octet hexa en décimal }
FUNCTION PuissanceDeux(No : Byte) : Byte;
 { Élève No à la puissance 2 }
PROCEDURE CLI;
 { Interdit les INTs }
PROCEDURE STI;
 { Autorise les INTs }

IMPLEMENTATION

 FUNCTION At : BOOLEAN;
 BEGIN
   At := (Mem[$F000:$FFFE] = $FC);
 END;
```

❷

```
FUNCTION Boole(Posit : Byte) : STRING;
CONST Reponse : ARRAY[0..1] OF STRING = ('Non', 'Oui');
BEGIN
  Boole := Reponse[Posit];
END;

FUNCTION ValeurNum(VAR Nb : STRING; Base : Byte) : Word;
CONST TabNos : ARRAY[0..15] OF Byte =
              (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
VAR No, Res, B : Word;
    i          : Byte;
BEGIN
  i := Length(Nb); Res := 0; B := 1;
  WHILE (i > 0) DO
  BEGIN
    No := TabNos[Ord(Nb[i])-48];
    No := No * B;
    B  := B*Base;
    Inc(Res, No);
    Dec(i);
  END;
  ValeurNum := Res;
END;

FUNCTION OctetDecVersBin(Octet : Byte) : STRING;
VAR  Dividende, Reste : Byte;
     MotifBin, MB : STRING;
BEGIN
  MotifBin := ''; MB := ''; Dividende := Octet;
  REPEAT
    Reste := (Dividende MOD 2);
    Str(Reste, MB);
    Dividende := (Dividende DIV 2);
    MotifBin := MB + MotifBin;
  UNTIL (Dividende = 0);
  IF (Length(MotifBin) < 8) THEN
  BEGIN
    REPEAT
      MotifBin := '0' + MotifBin;
    UNTIL (Length(MotifBin) = 8);
  END;
  OctetDecVersBin := MotifBin;
END;
```

❸

```
FUNCTION MotDecVersBin(Mot : Word) : STRING;
BEGIN
  MotDecVersBin := OctetDecVersBin(Hi(Mot)) + ' ' +
                   OctetDecVersBin(Lo(Mot));
END;

FUNCTION OctetDecVersHex(Octet : Byte) : STRING;
CONST TabHexa : ARRAY[0..15] OF Char =
                              '0123456789ABCDEF';
BEGIN
  OctetDecVersHex := TabHexa[(Octet AND $F0) SHR 4] +
                     TabHexa[(Octet AND $0F)];
END;

FUNCTION MotDecVersHex(Mot : Word) : STRING;
BEGIN
  MotDecVersHex := OctetDecVersHex(Hi(Mot)) +
                   OctetDecVersHex(Lo(Mot));
END;

FUNCTION HexaVersDecimal(Mot : STRING) : Word;
VAR i : Integer;
    No : Word;
BEGIN
  No := 0;
  FOR i := 1 TO Length(Mot) DO
    IF (Ord(Mot[i]) - 48 > 9) THEN
      No := (No SHL 4) + (Ord(Mot[i]) - 55)
    ELSE
      No := (No SHL 4) + (Ord(Mot[i]) - 48);
  HexaVersDecimal := No;
END;

FUNCTION PuissanceDeux(No : Byte) : Byte;
BEGIN
  IF (No = 0) THEN
    PuissanceDeux := 1
  ELSE
    PuissanceDeux := (2 * PuissanceDeux(No - 1));
END;

PROCEDURE CLI;
BEGIN
  INLINE($FA);
END;
```

❹

```
 PROCEDURE STI;
 BEGIN
   INLINE($FB);
 END;

END.
```

Unité FHandle.Tpu

❶

```
UNIT FHandle;                          { FHandle.Pas }

INTERFACE

USES Dos;

CONST                  { Dispositifs standard d'E/S }
 InputDos  : Word = 0;               { Con }
 OutputDos : Word = 1;               { Con }
 ErrDos    : Word = 2;               { Con }
 AuxDos    : Word = 3;               { Aux  = Com1 }
 PrnDos    : Word = 4;               { Prn  = Lpt1 }

TYPE
  Handle       = Word;
  StrAsciiZ    = ARRAY[0..255] OF Char;
  RecSearchFic = RECORD
                   Reserve      : ARRAY[0..$14] OF Byte;
                   Attr         : Byte;
                   Heure, Date  : Word;
                   Taille       : LongInt;
                   NomExt       : ARRAY[0..11] OF Char;
                 END;
  Tab          = Pointer;

VAR Erreur, ErreurDos : Word;
    RecSearch         : RecSearchFic;
```

❷

```
{ Renvoie le handle du fichier créé de nom "Nom" }
FUNCTION CreeFichier(Attr : Byte;
                     VAR Nom : StrAsciiZ) : Handle;

{ Ouvre le fichier de nom "Nom" dans le mode spécifié }
{ et renvoie son handle }
FUNCTION OuvreFichier(Mode : Byte;
                      VAR Nom : StrAsciiZ) : Handle;

{ Ferme le fichier de handle spécifié }
PROCEDURE FermeFichier(VAR h : Handle);

{Lit "Nb" octets à la position courante du pointeur dans }
{le fichier de handle "h", et les transfère dans le buffer}
{Renvoie le nombre d'octets réellement transférés }
FUNCTION LitFichierLogique(h : Handle; Nb : Word;
                           VAR Buf : Tab) : Word;

{ Ecrit "Nb" octets à la position courante du pointeur }
{ dans le fichier de handle "h" à partir du buffer. }
{ Renvoie le nombre d'octets réellement transférés  }
FUNCTION EcritFichierLogique(h : Handle; Nb : Word;
                             VAR Buf : Tab) : Word;

{ Efface le fichier de nom "Nom", renvoie 0 ou un code }
{ d'erreur }
FUNCTION EffaceFic(VAR Nom : StrAsciiZ) : Word;

{ Positionne le pointeur du fichier de handle "h" à la }
{ position "Position", selon la méthode "Methode".     }
{ Renvoie la position du pointeur de fichier           }
FUNCTION FSeek(Methode : Byte; h : Handle;
               Position : LongInt) : LongInt;

{ Modifie l'attribut du fichier de nom "Nom". Renvoie 0 }
{ ou un code d'erreur }
FUNCTION ChangeAttrFic(Attr : Byte;
                       VAR Nom : StrAsciiZ) : Word;

{ Renvoie l'attribut actuel du fichier de nom "Nom" }
FUNCTION DonneAttrFic(VAR Nom : StrAsciiZ) : Word;

{ Duplique le handle "h", et renvoie le nouveau handle }
FUNCTION NewHandle(h : Handle) : Word;
```

❸

```
{ Redirige le handle "hSource" vers le handle "hDest" }
FUNCTION RedirigeHandle(VAR hSrce, hDest : Handle) : Word;

{ Fait de "Rec" la DTA courante }
PROCEDURE SetDTA(VAR Rec : RecSearchFic);

{ Trouve le premier fichier correspondant à la chaîne }
{ de recherche et aux attributs. Renvoie 0 ou un code }
{ d'erreur }
FUNCTION FindFirstFic(Attr : Byte;
                      VAR Nom : StrAsciiZ) : Word;

{ Trouve le fichier suivant. Renvoie 0 ou un code }
{ d'erreur }
FUNCTION FindNextFic : Word;

{ Renomme le fichier "NomActuel" en "NouveauNom" }
FUNCTION RenFic(VAR NomActuel,
                NouveauNom : StrAsciiZ) : Word;

{ Change la date et l'heure du fichier de handle "h" }
FUNCTION ChangeDateHeure(h : Handle; Date,
                         Heure : Word) : Word;

{ Renvoie la date et l'heure du fichier de handle "h" }
FUNCTION DonneDateHeure(h : Handle; VAR Date,
                        Heure : Word) : Word;

{ Crée un fichier temporaire d'attribut "Attr" }
FUNCTION CreeTemporaire(Attr : Byte;
                        VAR Nom : StrAsciiZ) : Word;

{ Crée un fichier, et renvoie un code d'erreur s'il }
{ existe déjà }
FUNCTION CreeNouveauFic(Attr : Byte;
                        VAR Nom : StrAsciiZ) : Word;

{ Fixe le nombre maximum de handles auquel le programme }
{ a droit }
FUNCTION FixeMaxHandles(Nb : Word) : Word;

{ Vide le buffer du fichier de handle "h" }
FUNCTION FlusheBufferFic(h : Handle) : Word;

IMPLEMENTATION
```

❹

```
FUNCTION CreeFichier(Attr : Byte;
                     VAR Nom : StrAsciiZ) : Handle;
 VAR Regs : Registers;
 BEGIN
   WITH Regs DO
   BEGIN
     Ah := $3C; Cx := (0 SHL 8) + Attr;
     Ds := Seg(Nom); Dx := Ofs(Nom);
     MsDos(Regs);
     IF (Flags AND 1 = 1) THEN
     BEGIN
       Erreur := Ax; CreeFichier := $FFFF;
     END
     ELSE
     BEGIN
       Erreur := 0; CreeFichier := Ax;
     END;
   END;
 END;

FUNCTION OuvreFichier(Mode : Byte;
                      VAR Nom : StrAsciiZ) : Handle;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $3D; Al := Mode;
    Ds := Seg(Nom); Dx := Ofs(Nom);
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; OuvreFichier := $FFFF;
    END
    ELSE
    BEGIN
      Erreur := 0; OuvreFichier := Ax;
    END
  END;
END;

PROCEDURE FermeFichier(VAR h : Handle);
VAR Regs : Registers;
BEGIN
  WITH Regs DO
```

❺

```
   BEGIN
     Ah := $3E; Bx := h;
     MsDos(Regs);
     IF (Flags AND 1 = 1) THEN
       Erreur := Ax
     ELSE
       Erreur := 0;
   END;
END;

FUNCTION LitFichierLogique(h : Handle; Nb : Word;
                           VAR Buf : Tab) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $3F; Bx := h; Cx := Nb;
    Ds := Seg(Buf^); Dx := Ofs(Buf^);
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax;
      LitFichierLogique := 0;
    END
    ELSE
    BEGIN
      Erreur := 0;
      LitFichierLogique := Ax;
    END;
  END;
END;

FUNCTION EcritFichierLogique(h : Handle; Nb : Word;
                             VAR Buf : Tab) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $40; Bx := h; Cx := Nb;
    Ds := Seg(Buf^); Dx := Ofs(Buf^);
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
```

❻

```
    BEGIN
      Erreur := Ax;
      EcritFichierLogique := 0;
    END
    ELSE
    BEGIN
      Erreur := 0;
      EcritFichierLogique := Ax;
    END;
  END;
END;

FUNCTION EffaceFic(VAR Nom : StrAsciiZ) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $41; Ds := Seg(Nom); Dx := Ofs(Nom);
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax;
      EffaceFic := Ax;
    END
    ELSE
    BEGIN
      Erreur := 0;
      EffaceFic := 0;
    END;
  END;
END;

FUNCTION FSeek(Methode : Byte; h : Handle;
               Position : LongInt) : LongInt;
VAR Regs : Registers;
    Res  : LongInt;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $42; Al := Methode; Bx := h;
    Cx := Position AND $0000FFFF;
    Dx := Position AND $FFFF0000;
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
```

❼

```
    BEGIN
      Erreur := Ax; FSeek := $FFFFFFF;
    END
    ELSE
    BEGIN
      Erreur := 0; Res := (Dx SHL 16) + Ax;
      FSeek := Res;
    END;
  END;
END;

FUNCTION ChangeAttrFic(Attr : Byte;
                       VAR Nom : StrAsciiZ) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $43; Al := 1; Ch := 0; Cl := Attr;
    Ds := Seg(Nom); Dx := Ofs(Nom);
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; ChangeAttrFic := $FFFF;
    END
    ELSE
    BEGIN
      Erreur := 0; ChangeAttrFic := Cx;
    END;
  END;
END;

FUNCTION DonneAttrFic(VAR Nom : StrAsciiZ) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $43; Al := 0; Ds := Seg(Nom); Dx := Ofs(Nom);
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; DonneAttrFic := $FFFF
    END
    ELSE
```

❽

```
    BEGIN
      Erreur := 0; DonneAttrFic := Cx;
    END;
  END;
END;

FUNCTION NewHandle(h : Handle) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $45; Bx := h;
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; NewHandle := $FFFF;
    END
    ELSE
    BEGIN
      Erreur := 0; NewHandle := Ax;
    END;
  END;
END;

FUNCTION RedirigeHandle(VAR hSrce,
                        hDest : Handle) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $46; Bx := hDest; Cx := hSrce;
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax;
      RedirigeHandle := $FFFF;
    END
    ELSE
    BEGIN
      Erreur := 0;
      RedirigeHandle := 0;
    END;
  END;
END;
```

❾

```
PROCEDURE SetDTA(VAR Rec : RecSearchFic);
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $1A; Ds := Seg(Rec); Dx := Ofs(Rec);
    MsDos(Regs);
  END;
END;

FUNCTION FindFirstFic(Attr : Byte;
                      VAR Nom : StrAsciiZ) : Word;
VAR Regs : Registers;
BEGIN
  SetDTA(RecSearch);
  WITH Regs DO
  BEGIN
    Ah := $4E; Ch := 0; Cl := Attr; Ds := Seg(Nom);
    Dx := Ofs(Nom);
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; FindFirstFic := Ax;
    END
    ELSE
    BEGIN
      Erreur := 0; FindFirstFic := 0;
    END;
  END;
END;

FUNCTION FindNextFic : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $4F;
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      ErreurDos := Ax;
      FindNextFic := 0;
    END
    ELSE
```

10

```
    BEGIN
      ErreurDos := 0;
      FindNextFic := 0;
    END;
  END;
END;

FUNCTION RenFic(VAR NomActuel,
                NouveauNom : StrAsciiZ) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ds := Seg(NomActuel); Dx := Ofs(NomActuel);
    Es := Seg(NouveauNom); Di := Ofs(NouveauNom);
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; RenFic := Erreur;
    END
    ELSE
    BEGIN
      Erreur := 0; RenFic := 0;
    END;
  END;
END;

FUNCTION ChangeDateHeure(h : Handle; Date, Heure : Word) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $57; Al := 1; Bx := h; Cx := Heure; Dx := Date;
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; ChangeDateHeure := Ax;
    END
    ELSE
    BEGIN
      Erreur := 0; ChangeDateHeure := 0;
    END;
  END;
END;
```

11

```
FUNCTION DonneDateHeure(h : Handle; VAR Date,
                        Heure : Word) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $57; Al := 0; Bx := h;
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; Date := 0; Heure := 0;
      DonneDateHeure := Ax;
    END
    ELSE
    BEGIN
      Erreur := 0; Date := Dx; Heure := Cx;
      DonneDateHeure := 0;
    END;
  END;
END;

FUNCTION CreeTemporaire(Attr : Byte;
                        VAR Nom : StrAsciiZ) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $5A; Ch := 0; Cl := Attr;
    Ds := Seg(Nom); Dx := Ofs(Nom);
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; CreeTemporaire := $FFFF;
    END
    ELSE
    BEGIN
      Erreur := 0; CreeTemporaire := Ax;
    END;
  END;
END;

FUNCTION CreeNouveauFic(Attr : Byte;
                        VAR Nom : StrAsciiZ) : Word;
VAR Regs : Registers;
```

```
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $5B; Ch := 0; Cl := Attr;
    Ds := Seg(Nom); Dx := Ofs(Nom);
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; CreeNouveauFic := $FFFF;
    END
    ELSE
    BEGIN
      Erreur := 0; CreeNouveauFic := Ax;
    END;
  END;
END;

FUNCTION FixeMaxHandles(Nb : Word) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $67; Bx := Nb;
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
    BEGIN
      Erreur := Ax; FixeMaxHandles := Ax;
    END
    ELSE
    BEGIN
      Erreur := 0; FixeMaxHandles := 0;
    END;
  END;
END;

FUNCTION FlusheBufferFic(h : Handle) : Word;
VAR Regs : Registers;
BEGIN
  WITH Regs DO
  BEGIN
    Ah := $68; Bx := h;
    MsDos(Regs);
    IF (Flags AND 1 = 1) THEN
```

13

```
    BEGIN
      Erreur := Ax; FlusheBufferFic := Ax;
    END
    ELSE
    BEGIN
      Erreur := 0; FlusheBufferFic := 0;
    END;
  END;
END;

END.
```

Annexe 2

Interruptions et fonctions cachées du DOS

De nombreuses fonctions de l'Int 21h ne sont pas documentées par Microsoft. Certaines d'entre elles sont pourtant essentielles au bon fonctionnement du DOS, et en tous les cas à la programmation système. Cette annexe passe donc toutes les fonctions cachées de l'Int 21h et toutes les interruptions non documentées du DOS en revue.

Interruptions DOS non documentées

Les *interruptions* non documentées utilisables par le programmeur système sont au nombre de trois. Ce sont :

1. Int 28h, DOS Safe Interrupt,
2. Int 29h, sortie sur le périphérique CON,
3. Int 2Eh, appeler COMMAND.COM.

En outre, les Ints 2Ah à 2Dh sont utilisées par le DOS lui-même comme routines internes. Leurs vecteurs pointent sur un code IRET. Les Int 30h à FFh servent généralement au matériel : ainsi l'Int 33h est réservée à la souris et l'Int 67h à la LIM EMS.

Int 28h : DOS Safe Interrupt

Le DOS utilise l'Int 28h en interne lorsqu'il lit les caractères entrés au clavier pour savoir s'il lui est possible d'utiliser les fonctions 0Ch et suivantes de l'Int 21h.

Les programmes résidents, et parmi eux, PRINT.COM du DOS, utilisent également cette interruption. Lorsqu'un programme attend qu'une touche soit pressée au clavier, le DOS arrête d'appeler l'Int 28h et signale ainsi aux autres applications qui pourraient être chargées en mémoire que les fonctions de numéro supérieur ou égal à 0Ch de l'Int 21h sont indisponibles. Lorsqu'une touche a été pressée et que la routine est terminée, l'Int 28h est appelée pour signaler qu'il est possible d'utiliser toutes les fonctions de l'Int 21h.

L'Int 28h est généralement utilisée avec la fonction 34h de l'Int 21h (connaître l'état du drapeau d'occupation du DOS). Le DOS appelle ou arrête d'appeler l'Int 28h et le programme résident appelle la fonction 34h de l'Int 21h en vue de déterminer l'état exact du système.

Int 29h : sortie d'un caractère sur le périphérique CON

Lorsqu'on exécute cette interruption, on est certain d'afficher le caractère en AL sur l'écran. Elle peut donc être particulièrement utile lorsque le programme peut être redirigé vers un fichier. Imaginons que vous souhaitiez afficher un écran d'aide dans un tel programme : si vous n'utilisez pas cette interruption, vous pouvez être certain que votre écran d'aide se retrouvera dans le fichier où les sorties ont été redirigées. En revanche, si vous faites appel à l'Int 29h, le texte d'aide sera bien affiché sur l'écran, *malgré la redirection.*

Int 2Eh : retourner à COMMAND.COM

Il est possible d'exécuter un ordre DOS sans passer par la fonction EXEC. On utilise pour cela l'Int 2Eh. Celle-ci appelle COMMAND.COM et lui passe en paramètre l'ordre à exécuter. Il s'agit d'une méthode bien plus rapide que la fonction EXEC, mais qui est loin d'être conseillée.

Si l'on souhaite l'utiliser, il faut :

1. réduire la taille mémoire occupée par l'appelant à l'aide de la fonction 4Ah ;
2. faire pointer DS:SI sur la chaîne de paramètres à passer en ligne de commande ;
3. exécuter l'Int 2Eh ;
4. réinitialiser la pile (SS et SP sont détruits par l'Int 2Eh).

La chaîne de paramètres a le format suivant :

Adresse	*Contenu*
00h	Taille de la chaîne (Return compris)
..	Chaîne de caractères terminée par Return.

Tableau A2.1
Format de la chaîne de paramètres de l'interruption 2Eh.

Malheureusement, l'Int 2Eh ne sauvegarde pas les valeurs des registres SS et SP : il faut donc réinitialiser la pile après avoir appelé cette interruption. C'est la principale raison pour laquelle il est déconseillé de faire appel à l'Int 2Eh. L'autre étant qu'elle n'est pas sûre d'emploi lorsque l'on souhaite exécuter un programme externe à COMMAND.COM, dont le chargement nécessite la création d'un PSP.

On peut cependant imaginer d'écrire un programme tirant parti de cette interruption, à condition qu'il sauvegarde les valeurs des registres SS et SP dans

une variable juste avant d'y faire appel et qu'il ne fasse appel qu'à des ordres internes à COMMAND.COM (comme DIR, CD, etc.).

Fonctions cachées de l'Int 21h

Certaines fonctions de l'Int 21h sont dites "réservées". Microsoft ne les a pas documentées : cela tient parfois à ce qu'elles sont purement et simplement inutiles, mais dans la plupart des cas, il n'y a pas de bonnes raisons à cet état de fait, sinon qu'elles pourraient ne pas être compatibles avec les versions à venir du DOS. Même dans ce cas, il est intéressant de les regarder et de les tester : comment retrouver un device driver ou savoir que le DOS ne doit pas être interrompu sans passer par elles ?

On aura donc tout intérêt à lire attentivement ce qui suit et à en tester le bon fonctionnement.

Fonctions 18h, 1Dh, 1Eh, 20h : fonctions DOS inutilisées

Ces numéros de fonctions ont été laissés disponibles pour une éventuelle compatibilité ascendante avec CP/M. Si Digital Research avait créé de nouvelles fonctions dans son système d'exploitation, la version suivante du DOS aurait compris ces nouvelles fonctionnalités. Comme chacun sait, ce ne fut pas le cas...

Fonction 1Fh : trouver le bloc de paramètres disque du périphérique par défaut

Cette fonction renvoie en DS:BX un pointeur sur l'enregistrement de la liste des blocs de paramètres disque concernant le périphérique par défaut. Lorsqu'on appelle cette fonction avec AH = 1Fh, celle-ci exécute la fonction 32h de l'Int 21h en lui passant DL égal à 0. Le *tableau A2.1* donne le format d'un bloc de paramètres disque. La liste des blocs de paramètres contient autant d'enregistrements que le PC comporte de disques logiques.

Adresse	*Description*
00h	Lecteur (0 = A:, 1 = B:, etc.)
01h	Sous-unité dans le driver
02h	Octets par secteur
04h	Secteurs - 1 par cluster
05h	Masque de conversion secteur vers cluster `NoClust := (NoSect SHR BlocParam[5])`
06h	Nombre de secteurs réservés
08h	Nombre de FAT
09h	Nombre d'entrées du répertoire racine
0Bh	Numéro de secteur du cluster 2 (premier cluster de données)
0Dh	Dernier numéro de cluster
0Fh	Nombre de secteurs par FAT
10h	Numéro du premier secteur du répertoire racine
12h	Offset de l'en-tête du device driver
14h	Segment de l'en-tête du device driver
16h	ID Média
17h	00h ou FFh selon que l'on a déjà accédé au disque ou non
18h	Offset du prochain bloc de paramètres disque (FFFFh si c'est le dernier)
1Ah	Segment du prochain bloc de paramètres disque (FFFFh si c'est le dernier)

Tableau A2.2
Format du bloc de paramètres disque.

Fonction 32h : trouver le bloc de paramètres disque du lecteur spécifié

Au contraire de la fonction 1Fh, la fonction 32h accepte en paramètre le numéro de lecteur, qu'on lui passe en DL. Si le numéro de lecteur spécifié est invalide, elle retourne FFh en AL. Si tout s'est bien passé, AL contient 00h en sortie. DS:BX contient un pointeur sur l'enregistrement du bloc de paramètres disque concernant le lecteur spécifié.

Fonction 34h : lire le drapeau d'occupation du DOS

La fonction 34h renvoie en ES:BX un pointeur sur le drapeau d'occupation du DOS. Le drapeau d'occupation du DOS est un octet mis à zéro lorsqu'il est possible d'interrompre le DOS, et différent de zéro dans le cas contraire. On utilisera avec profit cette fonction, ainsi que l'interruption 28h, lors de la programmation de programmes résidents.

Fonction 37h : lire/écrire le caractère de switch

Cette fonction modifie le caractère de switch utilisé dans les lignes de commandes du DOS. Il s'agit habituellement du caractère d'anti-slash ('\'). Si vous préférez utiliser le signe de division ('/'), ou n'importe quel autre caractère facile à taper, il suffit d'exécuter la fonction 37h de l'Int 21h en mettant AL à 1 (écriture du caractère de switch) et en passant le code ASCII du caractère désiré en DL. Si la fonction s'est bien déroulée, AL doit être différent de FFh et DL doit contenir le code ASCII que vous lui avez passé en entrée.

Fonction 50h : installer le segment de PSP

Lorsqu'on appelle la fonction 50h avec le registre BX contenant le segment d'un PSP déjà créé, elle installe ce PSP comme PSP courant du programme à lancer. Celui-ci peut être un résident : c'est du reste l'intérêt principal de cette fonction que de permettre à un TSR d'installer son propre PSP. Cela lui assure notamment que les handles des fichiers qu'il ouvrira seront bien sauvegardés dans sa propre table des handles, et non dans celle du PSP du programme qui a été interrompu par le TSR.

Fonction 51h : obtenir l'adresse du PSP courant

La fonction 51h de l'Int 21h retourne en BX l'adresse du PSP courant. On l'utilise généralement en liaison avec la fonction 50h dans un programme résident (TSR), de façon à sauvegarder l'adresse du PSP du programme interrompu avant que le résident n'active son propre PSP. Cela permet de réinstaller le PSP du programme interrompu en tant que PSP courant lorsque le TSR rend la main.

Fonction 52h : obtenir l'adresse du nœud d'informations du DOS

La fonction 52h est probablement la plus importante des fonctions réservées de l'Int 21h du DOS. Elle permet en effet d'accéder au nœud d'informations du DOS, qui est entre autres la structure chargée de gérer les MCB, les device drivers, la SFT, etc.

Adresse	Description
ES:BX-04h	Offset du premier MCB
ES:BX-02h	Segment du premier MCB
ES:BX	Offset du premier bloc de paramètres disque
ES:BX+02h	Segment du premier bloc de paramètres disque
ES:BX+04h	Offset de la SFT handles
ES:BX+06h	Segment de la SFT handles
ES:BX+08h	Offset du device driver CLOCK$
ES:BX+0Ah	Segment du device driver CLOCK$
ES:BX+0Ch	Offset du device driver CON
ES:BX+0Eh	Segment du device driver CON
ES:BX+10h	Taille maximum d'un secteur
ES:BX+12h	Offset du premier buffer disque
ES:BX+14h	Segment du premier buffer disque
ES:BX+16h	Offset de la table des chemins
ES:BX+18h	Segment de la table des chemins
ES:BX+1ah	Offset de la SFT par FCB
ES:BX+1ch	Segment de la SFT par FCB
ES:BX+1eh	Taille de la SFT par FCB
ES:BX+20h	Nombre de drives
ES:BX+21h	Numéro du dernier drive
(le driver NUL commence ici)	
ES:BX+22h	Offset du prochain en-tête de device driver
ES:BX+24h	Segment du prochain en-tête de device driver
ES:BX+26h	Attribut du driver NUL (= 8004h)
ES:BX+28h	Offset de la routine de stratégie du driver NUL
ES:BX+2ah	Offset de la routine d'interruption du driver NUL
ES:BX+2ch	Nom du driver nul = 'NUL

Tableau A2.3
Format du nœud d'informations du DOS.

Le nœud d'informations du DOS est une structure de données non documentée par Microsoft. Comme telle, elle n'est pas supposée avoir un format fixe d'une version l'autre. Le *tableau A2.2* est donc valable pour le DOS IBM 3.3 et pour le DOS Microsoft 3.3. Les autres versions (autres numéros ou autres constructeurs) devront faire l'objet d'un examen attentif en fonction des renseignements fournis ici. Ce sont essentiellement les adresses auxquelles se trouvent les renseignements qui peuvent varier. La nature de ces renseignements est, elle, fixe. Voyez le programme `InfoDos.Pas` pour savoir comment tester la fonction `52h` de l'Int `21h`.

Listing A2.4
Programme InfoDos.Pas. ❶

```
PROGRAM VerifieNoeudInfo;  { InfoDos.Pas }

USES Dos, Sys;

TYPE
  Dev      =  RECORD
                NextHeaderOfs,
                NextHeaderSeg,
                Attribut,
                StrategyOfs,
                InterruptOfs    : Word;
                Nom             : ARRAY[0..7] Of Char;
              END;
   Noeud   =  RECORD
                 McbOfs, McbSeg,
                 DcbOfs, DcbSeg,
                 HdlSFTOfs, HdlSFTSeg,
                 ClockOfs, ClockSeg,
                 ConOfs, ConSeg,
                 MaxSect,
                 BufOfs, BufSeg,
                 CDSOfs, CDSSeg,
                 FcbSftOfs, FcbSftSeg,
                 FcbSftTail,
                 NbDrv, LastDrv          : Word;
                 NUL                     : Dev;
              END;

VAR  Info : Noeud;
     Ok   : BOOLEAN;

FUNCTION LitInfoDos : BOOLEAN;
VAR Regs : Registers;
```

➥

Programme InfoDos.Pas. ❷

```
Begin
  WITH Regs DO
  BEGIN
    Ah := $52;
    MsDos(Regs);
    IF (FLAGS AND 1 = 1) THEN
      LitInfoDos := FALSE
    ELSE
    BEGIN
      WITH Info DO
      BEGIN
        McbOfs := MemW[Es:(Bx-4)];
        McbSeg := MemW[Es:(Bx-2)];
        DcbOfs := MemW[Es:Bx];
        DcbSeg := MemW[Es:Bx+2];
        HdlSFTOfs := MemW[Es:Bx+4];
        HdlSFTSeg := MemW[Es:Bx+6];
        ClockOfs := MemW[Es:Bx+8];
        ClockSeg := MemW[Es:Bx+$A];
        ConOfs := MemW[Es:Bx+$C];
        ConSeg := MemW[Es:Bx+$E];
        MaxSect := MemW[Es:Bx+$10];
        BufOfs := MemW[Es:Bx+$12];
        BufSeg := MemW[Es:Bx+$14];
        CDSOfs := MemW[Es:Bx+$16];
        CDSSeg := MemW[Es:Bx+$18];
        FcbSftOfs := MemW[Es:Bx+$1A];
        FcbSftSeg := MemW[Es:Bx+$1C];
        FcbSftTail := MemW[Es:Bx+$1E];
        NbDrv := Mem[Es:Bx+$20];
        LastDrv := Mem[Es:Bx+$21];
        WITH NUL DO
        BEGIN
          NextHeaderOfs := MemW[Es:Bx+$22];
          NextHeaderSeg := MemW[Es:Bx+$24];
          Attribut      := MemW[Es:Bx+$26];
          StrategyOfs   := MemW[Es:Bx+$28];
          InterruptOfs  := MemW[Es:Bx+$2A];
          Move(Mem[Es:Bx+$2C], Nom, 8);
        END;
      END;
      LitInfoDos := TRUE;
    END;
  END;
END;
```

➡

Programme InfoDos.Pas. ❸

```
Begin
  Ok := LitInfoDos;
  IF Ok THEN
  BEGIN
    WriteLn;
    WITH Info DO
    BEGIN
      WriteLn('Premier MCB en : ',
               MotDecVersHex(McbSeg),':',
               MotDecVersHex(McbOfs));
      WriteLn('DCB en : ', MotDecVersHex(DcbSeg), ':',
               MotDecVersHex(DcbOfs));
      WriteLn('SFT Handles en : ',
               MotDecVersHex(HdlSFTSeg), ':',
               MotDecVersHex(HdlSFTOfs) );
      WriteLn('CLOCK$ en : ', MotDecVersHex(ClockSeg),
              ':', MotDecVersHex(ClockOfs) );
      WriteLn('CON en : ', MotDecVersHex(ConSeg), ':',
              MotDecVersHex(ConOfs));
      WriteLn('longueur maximale d''un secteur : ',
              MaxSect);
      WriteLn('Buffers en : ',
              MotDecVersHex(BufSeg),':',
              MotDecVersHex(BufOfs));
      WriteLn('Table des chemins en : ',
              MotDecVersHex(CDSSeg), ':',
              MotDecVersHex(CDSOfs) );
      WriteLn('SFT FCB en : ',
              MotDecVersHex(FcbSftSeg),':',
              MotDecVersHex(FcbSftOfs));
      WriteLn('Taille de la SFT FCB = ',
              MotDecVersHex(FcbSftTail));
      WriteLn('Nbre de drives : ', NbDrv);
      WriteLn('Dernier drive : ', LastDrv);
      WITH NUL DO
      BEGIN
        WriteLn('Device =  ', Nom);
        WriteLn(#9,'Prochain Header de Device en ',
                MotDecVersHex(NextHeaderOfs),':',
                MotDecVersHex(NextHeaderSeg));
        WriteLn(#9,'Attribut   : ',
                MotDecVersHex(Attribut));
        WriteLn(#9,'Stratégie à l''Offset : ',
                MotDecVersHex(StrategyOfs));
```

➥

Programme InfoDos.Pas. ❹

```
        WriteLn(#9, 'Interruption à l''Offset : ',
                MotDecVersHex(InterruptOfs));
      END;
    END;
  END
  ELSE
  BEGIN
    WriteLn(' Abandon');
    Halt(1);
  END;
End.
```

Fonction 53h : conversion d'un BIOS Parameter Block en un bloc de paramètres disque

La fonction 53h recopie les informations utiles du BIOS Parameter Block d'un disque (passé en DS:SI) dans un bloc de paramètres disque dont on lui aura passé l'adresse en ES:BP. Cela correspond à peu près aux informations supplémentaires dont nous avons parlé au chapitre 6 (*Disques au niveau logique : structures DOS de bas niveau*).

Adresse	***Description***
ES : PB + 00h	Octets par secteur
ES : PB + 00h02h	Secteurs par cluster
ES : PB + 00h03h	Secteurs réservés
ES : PB + 00h05h	Nombre de FAT
ES : PB + 00h06h	Masque de décalage du numéro de cluster vers le numéro de secteur
ES : PB + 00h07h	Nombre d'entrées dans le répertoire racine
ES : PB + 00h09h	Nombre total de secteurs
ES : PB + 00h0Bh	ID Média
ES : PB + 00h0Ch	Nombre de secteurs par FAT
ES : PB + 00h0Eh-0Fh	Inconnu

Tableau A2.5
Le bloc de paramètres disque renvoyé par la fonction 53h.

Ici, comme pour la fonction 52h de l'Int 21h, il faudra prendre soin de vérifier le format du bloc de paramètres disque avant de l'utiliser dans un programme.

Fonction 55h : création d'un PSP

Contrairement à la fonction 26h (documentée) qui duplique le PSP courant, la fonction 55h crée un nouveau PSP à l'adresse contenue par le registre DX. En outre, elle met à jour le champ "propriétaire" du nouveau PSP.

Adresse	*Description*	*Taille*
00h	Int 20h (20CDh)	1 mot
02h	Dernier bloc mémoire alloué (exprimé en paragraphes)	1 mot
04h	*Inconnu (00h) ou numéro du PSP courant*	1 octet
05h	Appel long au DOS (9Ah F0h FEh 1Dh F0h)	5 octets
0Ah	Adresse de l'Int 22h	2 mots
0Eh	Adresse de l'Int 23h	2 mots
12h	Adresse de l'Int 24h	2 mots
16h	*Adresse de segment du PSP père*	1 mot
18h	*FHT (table des handles)*	20 octets
2Ch	Adresse de segment du bloc d'environnement	1 mot
2Eh	*SS:SP du programme lors d'une Int 21h (sauvegarde de la pile)*	2 mots
32h	*Taille de la FHT (0014h)*	1 mot
34h	*Adresse de la FHT (0018h XXXXh)*	2 mots
38h	*Adresse du prochain PSP (FFFFh FFFFh) inutilisé*	2 mots
3Ch	*Inconnu (20 x 00h) ou seconde FHT*	20 octets
50h	*Int 21h (21CDh)*	1 mot
52h	*RetF (CBh)*	1 octet
53h	*Inconnu (0000h)*	2 octets
55h	*Extension du premier FCB*	7 octets
5Ch	Premier FCB	16 octets
6Ch	Second FCB	16 octets
7Ch	*Inconnu (0000h:0000h) ou Adresse de la seconde FHT*	2 mots
80h	DTA par défaut (ligne de commande)	128 octets

Tableau A2.6
Format d'un PSP et de ses champs réservés.

Remarque — Les champs dont la description est en caractères italiques ne sont pas documentés par Microsoft. Ceux qui sont marqués "Inconnu" sont généralement remplis avec la valeur `00h`. Quant aux champs ayant une longueur de deux mots consécutifs, ils stockent une adresse sous la forme `Déplacement:Segment` : les mots sont donc à intervertir si l'on veut obtenir l'adresse exacte. L'adresse du prochain PSP (champ d'offset `38h`) n'est pratiquement jamais mise à jour par le DOS. En revanche, le numéro du PSP père (champ d'offset `16h`) est indiqué lorsque l'on emploie la fonction `55h` pour créer un PSP.

Fonction 60h : transformer un nom de chemin relatif en nom de chemin absolu

La fonction `60h` transforme le nom de chemin relatif qu'on lui a passé en `DS:SI` en un nom de chemin absolu, qu'elle renvoie en `ES:DI`. Attention, aucune vérification n'est faite, et le nom de chemin absolu renvoyé n'est jamais que celui en cours, étendu à la chaîne passée : c'est une simple opération de concaténation de chaînes qui est effectuée là.

Référence des interruptions et fonctions non documentées

Int numéro	*Entrées*	*Sorties*
21h	Ah := 1Fh	Ds := Seg(DskParamBloc) Bx := Ofs(DskParamBloc)
21h	Ah := 32h Dl := Lecteur (A: = 1)	Ds := Seg(DskParamBloc) Bx := Ofs(DskParamBloc) Al = FFh si erreur
21h	Ah := 34h	Es := Seg(FlagDosOccupé) Bx := Ofs(FlagDosOccupé)
21h	Ah := 37h Al = 0 -> Lire Al = 1 -> Ecrire	Dl := Caractère de switch Al = FFh si erreur
21h	Ah := 50h Bx := Adresse PSP	Rien

➥

(suite du tableau)

Int numéro	Entrées	Sorties
21h	Ah := 51h	Bx := Segment PSP courant
21h	Ah := 52h	Es := Seg(NoeudDosInfo) Bx-02h := Ofs(NoeudDosInfo)
21h	Ah := 53h Ds := Seg(BPB) Si := Ofs(BPB) Es := Seg(DskParamBloc) Bp := Ofs(DskParamBloc)	Es := Seg(DskParamBloc) Bp := Ofs(DskParamBloc)
21h	Ah := 55h Dx := Adresse PSP	Rien
21h	Ah := 60h Ds := Seg(Chemin) Si := Ofs(Chemin) Es := Seg(CheminAbs) Di := Ofs(CheminAbs)	Es := Seg(CheminAbs) Di := Ofs(CheminAbs)
28h	Rien	Rien
29h	Al := Caractère à afficher à l'écran	Rien
2Eh	Ds := Seg(Commande) Si := Ofs(Commande)	Rien

Tableau A2.7
Interruptions et fonctions réservées.

Dernières remarques

Ceux d'entre vous qui auront lu attentivement cette annexe n'auront pas manqué d'y remarquer l'absence de certaines fonctions non documentées du DOS : leur signification reste à découvrir.

Annexe 3

Bibliographie

Cette bibliographie n'est pas exhaustive, mais elle ne contient que des ouvrages que l'auteur a lu et auxquels il a jugé bon de se référer. De nombreux autres, non cités, ont été éliminés : ils contenaient des erreurs ou ne traitaient pas de programmation système.

Le matériel

Ouvrages théoriques

Architecture de l'ordinateur – Du circuit logique au logiciel de base
par Andrew Tanenbaum, InterEditions, Collection iia, 470 pages.

Un excellent livre sur le fonctionnement des ordinateurs, même s'il date un peu (les exemples concernent le PDP 11, l'IBM 370, le Z80, et le M 68 000). Tous les principes expliqués sont également valables sur PC. Le texte et les schémas sont très clairs.

Cours fondamental des microprocesseurs
par Henri Lilen, Editions Radio, 321 pages.

Si vous n'avez toujours pas compris comment fonctionne un microprocesseur après avoir lu ce livre, c'est à désespérer ! Il n'y a pas plus clair sur le sujet, ni plus pratique. Un bon complément du précédent, plus orienté vers la théorie.

Cours pratique de logique pour microprocesseurs
par Henri Lilen, Editions Radio, 254 pages.

Tout sur la logique booléenne et ses applications. Excellent.

Références constructeurs

iAPX 88 Book with an introduction to the iAPX 188 par Intel Corp., 182 pages.

80 286 and 80 287 Programmer's Reference Manual par Intel Corp., 200 pages.

80 386 DX Programmer's Reference Manual par Intel Corp., 230 pages.

80 286 Hardware Reference Manual par Intel Corp., 190 pages.

80 386 Hardware Reference Manual par Intel Corp., 200 pages.

Ce sont les livres de référence : les trois premiers présentent les modes d'adressage de la mémoire et les instructions du microprocesseur, les deux derniers sont plus attachés à son fonctionnement. Pas toujours faciles à lire,

et encore moins à manipuler, tant les pages sont fines. Merci au département "Litterature" d'Intel France, qui a bien voulu nous faire parvenir ces ouvrages.

WD37C65C Floppy Disk Subsystem Controller par Western Digital, 32 pages.

WD57C65 Floppy Disk Subsystem Controller par Western Digital, 43 pages.

WD50C12 Winchester Disk Controller par Western Digital, 43 pages.

WD42C22A Winchester Disk Subsystem Controller par Western Digital, 103 pages.

Il s'agit là des notices techniques de Western Digital sur leurs contrôleurs de disques et disquettes. Elles sont particulièrement utiles si l'on s'intéresse à la programmation matérielle (dont nous n'avons pas parlé). Nous remercions Winchester France de nous les avoir envoyées.

Ouvrage de référence

The IBM PC From the Inside Out, revised edition
par Murray Sargent III and Richard L. Shoemaker, Addison-Wesley Publishing Company, 468 pages.

On voit mal ce qu'il peut rester à écrire sur le sujet après ce livre. Tout le matériel de l'IBM PC et de l'IBM AT est présenté, du flip-flop au disque dur en passant par le contrôleur DMA. Les quatre premiers chapitres s'intéressent essentiellement à la programmation en Assembleur, les trois suivants plongent dans les entrailles du micro-ordinateur, les chapitres 8, 9 et 10 font le tour des périphériques et les deux derniers présentent l'environnement logiciel (DOS, compilateurs, éditeurs), les techniques de construction d'un micro et de débogage matériel. Complet, simple, bien conçu et efficace : que demander de plus ?

Le système d'exploitation

Ouvrages théoriques

Les systèmes d'exploitation, conception et mise en œuvre
par Andrew Tanenbaum, InterEditions, collection iia, 758 pages.

Ce gros livre est sans doute ce qu'il s'est fait de mieux sur le sujet : il présente tous les principes de fonctionnement des systèmes d'exploitation et contient le code source de Minix, un système multi-tâches multi-utilisateur

compatible Unix, écrit en C et en Assembleur par Tanenbaum pour les besoins de la cause. Exceptionnel de clarté.

Systèmes d'exploitation, concepts et algorithmes
par Joffroy Beauquier et Béatrice Bérard, Mc Graw-Hill, collection informatique, 534 pages.

Moins imposant que le précédent, ce livre-ci a l'intérêt de détailler les principaux algorithmes qui servent à la création des systèmes d'exploitation. Il a le défaut de les présenter sous une forme exclusivement mathématique et de ne pas fournir de code. Une bonne mise au point.

Référence constructeur

Microsoft MS-DOS v. 3.3 Programmer's Reference
par Microsoft Corp, 469 pages.

C'est la source principale des livres publiés par Duncan et Norton. Ils ont fait bien mieux.

Ouvrages de référence

The MS-DOS Encyclopedia, Complete and unabridged
General editor Ray Duncan, Penguin Books, Collection Microsoft Press, 1 530 pages.

Si vous n'en n'achetez qu'un, ce doit être celui-ci. Il contient *tous* les renseignements sur le DOS publiés par Microsoft. Tout y est détaillé, expliqué, commenté, programmé. Un seul défaut : il est lourd.

Advanced MS-DOS programming, second edition
par Ray Duncan, Penguin Books, Collection Microsoft Press, 646 pages.

Si vous n'achetez pas l'Encyclopédie, achetez au moins celui-ci et le suivant. Ce sont les meilleures références sur les fonctions du DOS et ses structures de données. Comparé au précédent, ses chapitres sont un peu courts, mais restent suffisamment clairs.

The New Peter Norton Programmer's Guide to the PC & PS/2
par Peter Norton and Richard Wilton, Penguin Books, Collection Microsoft Press, 470 pages.

Le complément parfait à *Advanced MS-DOS Programming*. Il couvre presque tous les aspects de la programmation système, dans les limites officielles fixées par Microsoft et IBM (pas question ici de SFT, FHT, MCB, etc.). Il a en outre l'avantage de s'étendre un peu plus sur le BIOS que l'ouvrage de Duncan.

Fonctionnement interne du DOS

The Waite Group's MS-DOS Papers for MS-DOS developers and power users, Howard W. Sams & Company, 562 pages.

Ce livre, écrit par plusieurs auteurs, a l'avantage de présenter et de mettre en œuvre des techniques avancées de programmation, comme l'écriture de device drivers ou la programmation du port série. Il s'intéresse également aux fonctions non documentées du DOS, mais éparpille les explications et les schémas à travers plusieurs chapitres. Un très bon livre, très clair, qui doit être lu attentivement.

The Waite Group's MS-DOS Developer's Guide, second edition,
Howard W. Sams & Company, 766 pages.

C'est le meilleur livre que l'auteur connaisse sur la programmation : il aborde tous les sujets de manière claire, contient nombre de schémas, offre d'excellents programmes assembleurs et C. Son seul défaut tient à ce qu'il a été co-écrit : certains schémas reparaissent sous diverses formes, et l'on ne sait plus où est le bon.

Le BIOS

Références constructeurs

System BIOS for IBM PC/XT/AT Computers and Compatibles, The Complete Guide to ROM-Based System Software,
Addison-Wesley Publishing Company, Collection Phoenix Technical Reference Series, 494 pages.

Incroyablement clair, très fourni en tableaux de toutes sortes, le ROM-BIOS au complet se trouve pour la première fois bien expliqué. Malheureusement, il n'est pas listé.

IBM Personnal Computer AT Technical Reference,
International Business Machines Corporation.

N'ayant pu disposer que d'une photocopie partielle, l'auteur ne peut pas juger de la qualité de cet ouvrage. Le code est en revanche très intéressant. Toute personne pouvant faire parvenir à l'auteur une photocopie complète est remerciée d'avance et sera remboursée.

L'Assembleur

The IBM PC From the Inside Out, revised edition,
par Murray Sargent III and Richard L. Shoemaker,
Addison-Wesley Publishing Company, 468 pages.

Les quatre premiers chapitres constituent l'une des meilleures introductions à l'Assembleur jamais publiée. Mais il ne s'agit pas d'un livre d'apprentissage à l'Assembleur. A compléter par l'un des suivants.

Peter Norton's Assembly Language Book for the IBM PC, revised and expanded, par Peter Norton and John Socha, Brady Books, 442 pages.

Un peu confus, mais bien fait, ce livre a l'avantage d'être construit à partir d'un exemple que l'on améliore au fur et à mesure que l'on découvre de nouvelles techniques. Attention aux erreurs de frappe que l'on trouve dans certains listings !

Turbo-Assembler v. 1.0, Le Manuel de l'utilisateur,
Borland, 598 pages.

Turbo-Assembler v. 2.0, User's Guide,
Borland, 483 pages.

Que vous ayez une version ou l'autre, faites l'effort d'en lire le manuel très bien fait, et qui vous apprendra l'Assembleur aussi bien que la plupart des livres.

Turbo Pascal

Le Désassembleur 8086 Colibri,
par John Colibri, Editions Mnémodyne, 290 pages.

Ce livre contient les sources d'un désassembleur de fichiers .COM écrit en Turbo-Pascal v. 3.0. Même s'il n'est pas à jour, c'est de loin le meilleur pour apprendre à programmer système.

Au cœur de Turbo-Pascal,
par John Colibri, Editions Mnémodyne, 484 pages.

Là encore, les renseignements que contient ce livre sont devenus obsolètes. Mais les techniques employées pour les obtenir et le désassemblage systématique, donneront d'heureuses idées à ceux qui s'intéressent à la fois au Pascal et à la programmation système.

Votre disquette d'accompagnement

Cette annexe donne la liste des programmes se trouvant sur votre disquette d'accompagnement et – à l'exception d'un – dans le livre. Il peut arriver que ceux-ci diffèrent quelque peu. Les divergences les plus fréquentes concernent les commentaires, le format des lignes de programmes (qui sont étalées sur plusieurs lignes dans le livre, mais pas dans les sources), l'emploi de majuscules pour distinguer les mots-clefs de Turbo Pascal des autres termes et, parfois, quelques tests. La plupart des tests qui apparaîssent dans les programmes de la disquette mais pas dans ceux du livre affichent un message d'erreur si l'utilisateur n'a pas donné de paramètres dans la ligne de commande.

Ces différences n'affectent pas le fonctionnement général du programme : lorsqu'il a fallu revoir un programme de fond en comble, la version définitive a bien entendu remplacé l'ancienne dans le corps même de l'ouvrage.

Enfin, certains programmes faisant appel aux fonctions non documentées du DOS peuvent se comporter bizarrement sur une machine et normalement sur une autre : les fonctions non documentées ne sont en effet pas portables. Le fonctionnement de ces programmes n'est assuré qu'avec un DOS 3.3 Microsoft ou IBM.

Titre	*Description*	*Numéro*	*Page*
`Absolute.Asm`	Accède en lecture ou en écriture à un secteur logique défini. Module .ASM lié à des programmes Pascal.	5.13	*176-178*
`BiosData.Pas`	Affiche l'état courant des données du BIOS et en interprète la signification.	(1)	
`BootCopy.Pas`	Copie du secteur de boot d'une disquette quelconque sur une autre disquette qui peut avoir un format différent.	6.8	*205-209*
`BootRec.Pas`	Affichage des données du secteur de boot d'un disque.	6.4	*195-198*
`CmosRam.Pas`	Détermine le format des lecteurs de disquettes en lisant la RAM CMOS.	4.6	*126*
`DmpMem.Pas`	Dumpe de la mémoire par blocs de 512 octets. Permet de modifier le contenu de la RAM. Affiche en hexa et en décimal.	3.22	*106-114*
`DskStrct.Asm`	Donne des informations sur le disque passé en paramètre : nombre de secteurs par cluster, nombre d'octets par secteur, nombre de clusters, identificateur OEM, etc.	5.12	*170-175*
`Dumper.Pas`	Dump secteur par secteur, du disque. Affichage hexa et décimal. Possibilités de modification.	5.16	*181-189*
`Entrees.Pas`	Lit les entrées fichiers d'un répertoire, les affiche et permet de naviguer entre les répertoires.	7.9	*248-260*
`FAT.Pas`	Affiche les numéros de cluster d'un fichier en lisant la FAT.	6.16	*216-224*
`FHandle.Pas`	Interface de l'unité `FHandle`, qui permet au Pascal de gérer les fichiers par handles.	8.10	*284-285*
`Fhandle.Tpu`	L'unité `FHandle` permet de gérer les fichiers par la méthode des handles, c'est-à-dire comme le DOS.	A1.2	*344-356*
`Filtre.Pas`	Cherche une chaîne de caractères passée en paramètres dans le fichier d'entrée (fichier ou clavier) et inscrit dans le fichier de sortie (fichier, écran, imprimante) l'adresse d'offset où il l'a trouvée.	8.18	*308-310*
`Format.Pas`	Formate physiquement une disquette en utilisant les fonctions de l'Int `13h` du BIOS.	4.14	*149-155*
`Header.Pas`	Affiche l'en-tête d'un fichier `.EXE`.	9.7	*317-321*
`HeureFic.Pas`	Lit l'heure et la date d'une entrée fichier, les décode et les affiche.	7.5	*241-242*

(1) `BiosData.Pas` n'est pas listé dans le livre : sa longueur (900 lignes) était trop importante pour cela.

➥

(suite du tableau descriptif de la disquette d'accompagnement)

Titre	Description	Numéro	Page
InfoDos.Pas	Affiche les valeurs des champs du nœud d'informations du DOS.	3.4	*57-60*
InfoDos.Pas	Affiche les valeurs des champs du nœud d'informations du DOS.	A2.4	*364-367*
InitFrmt.Pas	Initialise le buffer des champs d'adresse pour le formatage d'un disque fixe en fonction de son entrelacement.	4.10	*139-140*
Int25h.Pas	Programme démontrant les possibilités d'accès en lecture absolue du module Absolute.Asm.	5.14	*178-179*
ListerFh.Pas	Liste un fichier texte sur un périphérique : écran, fichier ou imprimante.	8.11	*287-289*
LitFHT.Pas	Affiche les entrées de la FHT courante.	8.13	*291*
MapBuf.Pas	Affiche les en-têtes des buffers disques.	3.15	*76-78*
MapDBP.Pas	Affiche certains champs des blocs de paramètres disque.	3.11	*70-72*
MapDrv.Pas	Affiche les en-têtes des device drivers et leur adresse en mémoire.	3.9	*65-68*
MapInt.Pas	Affiche les adresses des routines d'interruptions.	3.6	*61-62*
MapMCB.Pas	Interprète les MCB et en affiche l'adresse mémoire.	3.19	*87-92*
MapPath.Pas	Affiche la table des chemins du DOS.	3.13	*73-74*
MapPSP.Pas	Fournit la liste des PSP présents et affiche les champs de celui qu'a choisi l'utilisateur.	3.21	*95-103*
MemBios.Pas	Affiche la taille de la RAM obtenue en lisant la donnée BIOS qui se trouve à l'adresse 0040h:0013h.	1.6	*32*
MemInt.Pas	Affiche la taille de la RAM obtenue en exécutant une Int 12h.	1.7	*33*
Memoire.Pas	Illustre l'utilisation des fonctions DOS d'attribution mémoire à l'intérieur d'un programme Pascal.	3.17	*83-84*
ModExe.Pas	Modifie les champs MinAlloc et MaxAlloc d'un en-tête de fichier .EXE.	9.15	*336-339*
ParamDsk.Pas	Affiche les données du BIOS et la table des paramètres du disque fixe.	4.13	*143-148*
ParmDskt.Pas	Affiche les données du BIOS et la table des paramètres disquettes.	4.7	*127-132*

➥

(suite du tableau descriptif de la disquette d'accompagnement)

Titre	Description	Numéro	Page
MemBios.Asm	Affiche la taille de la RAM obtenue en lisant la donnée BIOS qui se trouve à l'adresse 0040h:0013h.	1.4	*30-31*
MemInt.Asm	Affiche la taille de la RAM obtenue en exécutant une Int 12h.	1.5	*31-32*
Reloc.Pas	Charge un fichier .EXE, lit les adresses à reloger, calcule les relogements, et affiche les adresses suivies de la valeur qu'elles auraient si elles étaient vraiment relogées.	9.13	*330-334*
Rien.Asm	Ne fait rien. Sert de démonstration à la partie consacrée aux fichiers .EXE.	9.1	*313*
Rien.Pas	Même chose que Rien.Asm.	9.2	*313*
SectPart.Pas	Affiche les données de la table de partition d'un disque dur, en les détaillant lecteur logique par lecteur logique.	6.20	*231-234*
SFT.Pas	Permet d'accéder à l'entrée SFT de votre choix.	8.17	*297-306*
Sys.Pas	L'unité Sys fournit les primitives nécessaires à la programmation système.	A1.1	*341-344*
UnDel.Pas	Récupère un fichier effacé sur disque dur. Attention ! Ne fait pas suffisamment de vérifications. A n'utiliser que lorsqu'un seul fichier a été effacé dans le répertoire.	7.11	*264-272*
VirMem.Pas	Réduit la taille qu'il occupe en mémoire. Attention, VirMem ne fonctionne que lorsqu'il est compilé sur disque, et pas sous Turbo Pascal.	9.10	*324-325*

Index

G, H

I

L, M

N

P

R

S

T

U, V, Z



Achevé d'imprimer en novembre 1990
N° d'édition 86595-632-1 / N° d'impression L 36588
Dépôt légal novembre 1990 / Imprimé en France
ISBN : 2-86595-632-6

Découper ici avec un cutter

Sous ce rabat
VOTRE DISQUETTE
avec les programmes du livre



IMPORTANT :
Dupliquer cette disquette
avant toute utilisation,
et travailler avec la copie

Pour tout problème veuillez contacter notre Service Technique au (1) 47 40 66 42



Cet ouvrage s'adresse au programmeur système déjà familiarisé avec Turbo Pascal et l'Assembleur. Il y trouve d'abord une description soignée et détaillée des données du PC qui s'utilisent pour programmer directement la mémoire centrale, les disques et fichiers.

Il dispose, surtout, d'une grande variété de programmes montrant comment interpréter et exploiter ces données.

Pour une approche plus concrète de la programmation système sur PC

A l'appui des explications techniques : des tableaux précis, des schémas en grand nombre.

Encore plus d'exemples

Dans chaque chapitre, des exemples de programmes complets faisant appel aux fonctions les plus intéressantes du DOS et du BIOS.

Pour maîtriser plus vite

La disquette incluse sous cette couverture donne un accès immédiat aux programmes source du livre.

Thèmes abordés

- Les données du BIOS disponibles en RAM.
- La RAM gérée par les principales structures de données DOS.
- La gestion des disques au niveau physique par les fonctions BIOS.
- Le plan d'un disque et son organisation logique.
- Les structures DOS de bas niveau : secteur de boot, FAT et table de partition.
- Les structures DOS de haut niveau : répertoire racine, sous-répertoires et zone des données.
- Les fichiers de données et la gestion des handles par le DOS.
- Les fichiers .EXE : leur création et leur exécution.

Au terme de cet ouvrage, le programmeur dispose d'un bon entraînement à la programmation des structures internes du BIOS et du DOS. Il est capable de les mettre en œuvre dans ses propres programmes qu'il peut ainsi rendre encore plus rapides et plus efficaces.

Mémoires, disques et fichiers



ISBN : 2-86595-632-6
500762